TREND MICRO | research

Telfhash: An Algorithm That Finds Similar Malicious ELF Files Used in Linux IoT Malware

By Fernando Mercês (Threat Researcher)
and with contributions from Joey Costoya

The internet of things (IoT) has swiftly become a seemingly indispensable part of our daily lives. The IoT devices in pockets, homes, offices, cars, factories, and cities make people's lives more efficient and convenient. It is little wonder, then, that IoT adoption continues to increase. In 2019, the number of publicly known IoT platforms grew to 620, which was double the number of platforms in 2015. This year, 31 billion IoT devices are expected to be installed globally. Consequently, cybercriminals have been developing IoT malware such as backdoors and botnets for malicious purposes, including digital extortion. As reported in Trend Micro's latest annual security roundup, the number of brute-force logins made by IoT botnets in 2019 was triple the corresponding number in 2018.

Through the years, cybersecurity researchers have developed various helpful algorithms to identify large numbers of malicious files quickly and accurately as an effective measure in the fight against malware. But on the IoT front, as threats and attacks geared toward web-connected devices continue to grow exponentially, cybersecurity experts need to have a means to make their defensive measures systematic, accurate, and strong.

With the support of Trend Micro and my fellow cybersecurity researchers, I have created telfhash, an open-source clustering algorithm that effectively clusters malware targeting IoT devices running on Linux — i.e., Linux IoT malware — created using Executable and Linkable Format (ELF) files.

A historical look at how similar malware files are found

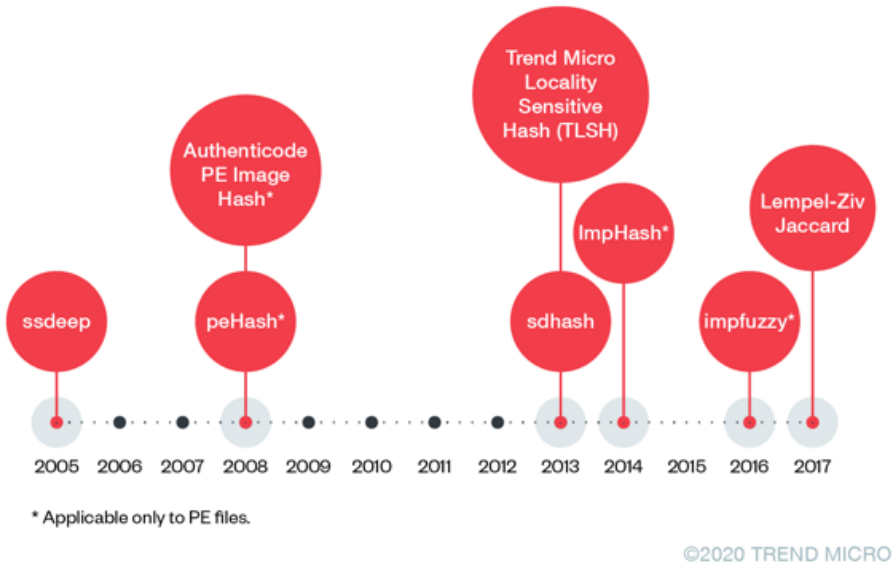

Figure 1. A timeline of algorithms created to find similar malware files

Several years ago, back when Windows was almost exclusively used as the default operating system in most users' machines, cybersecurity experts waged a war against malware, suspicious executables, and malicious scripts. In order to win the battle, they needed to find malicious files in a quick, effective, and automated manner.

Because of the massive use of Windows, the Windows Portable Executable (PE) was the format mainly used by virus writers and cybercriminal groups in creating malicious files. And in order to mitigate the spread of malicious PE files, malware researchers used several techniques to efficiently find similar PE files to investigate. These techniques also enabled researchers to better track cybercrime campaigns; once they found a single malicious file, which likely had a single set of command-and-control (C&C) servers, they could pivot on it to find other files belonging to the same campaign or cybercriminal group.

Different solutions appeared, like the regular hashing algorithms — such as MD5, SHA-1, and SHA-256 — that are applied to certain structures of a PE file. Another example of a good solution is peHash. The idea behind peHash is simple: Instead of calculating a hash value considering the whole PE file, only some of its structural parts (such as the file's field values and section headers) are examined. Some implementations of this hash function have been widely available for use.

Other researchers, meanwhile, developed similarity digests using Context Triggered Piecewise Hashes (CTPH) aka fuzzy hashes, which are able to identify known files that contain inserted, modified, or deleted data. These algorithms have two main features: They generate a hash, and they offer a comparison operation to determine similarity.

A widely used program for computing CTPH is ssdeep, with the help of its own library that is used to generate or compare fuzzy hashes, libfuzzy. Ssdeep, which is not limited to just executable files, examines the whole file — including all of its bytes — and generates the hashes of the most interesting or notable parts. Similar files would have a high probability to have similar but not identical hashes.

With cryptographic one-way hashes like MD5 or the SHA family, a single byte change could result in a totally different hash. This is because cryptographic hashes seek to preserve uniqueness (one to one), while similarity hashes like CTPH seek to preserve similarity (many to one). Similarity hashes provide a probabilistic number that represents similarity (usually between 0 and 100). To better illustrate how cryptographic and similarity hashing techniques work, we took six different samples from TheMoon malware, which targeted IoT devices in 2018. Despite each sample's having different MD5 values, the ssdeep hashes show similar values, as seen in the parts highlighted in yellow and blue in Figure 2.

Figure 2. A comparison between MD5 and ssdeep hashing techniques

The way ssdeep works involves hashing different pieces of a file and combining the results to generate a final hash value.

Over time, new hashing schemes have been developed. Similarity digest hash (sdhash), for instance, takes a totally different approach from ssdeep. It looks for sequences with the lowest probability of being randomly found in a file, hashes them, and places them into a probabilistic set representation called the

Bloom filter. It is safe to say that sdhash compensates for the accuracy and scalability limitations of ssdeep.

In an effort to deal with the shortcomings of both hashing techniques, Trend Micro created Trend Micro Locality Sensitive Hash (TLSH), a type of fuzzy hashing technique that highlights the locality-sensitive nature of a file instead of its similarity, and can be used in machine learning extensions of whitelisting. If a file is found to be similar to known legitimate files, it is deemed safe to run on a system. TLSH has proved that it can detect malware evasion techniques because sequence reordering does not change the hash. It is also a scalable tool with its fixed-length hash, which means that it is not dependent on the size of the input. In 2018, we used TLSH to analyze 2 million signed files to uncover a massive certificate signing abuse by a marketing adware plug-in called Browsefox.

Import hashing

I would like to highlight two solutions that are important in the development of telfhash. One of them is import hashing (ImpHash), which is primarily used in identifying malware binaries belonging to the same malware family. It analyzes similar malware files by getting the imported functions of a PE file (from the import directory) and its related library names, and creating a comma-separated list. Afterward, the list will be hashed using the MD5 checksum algorithm. In the example shown in Figure 3, we took a sample of Lokibot, a malware variant that is able to steal sensitive data from victim machines, to illustrate how ImpHash works.

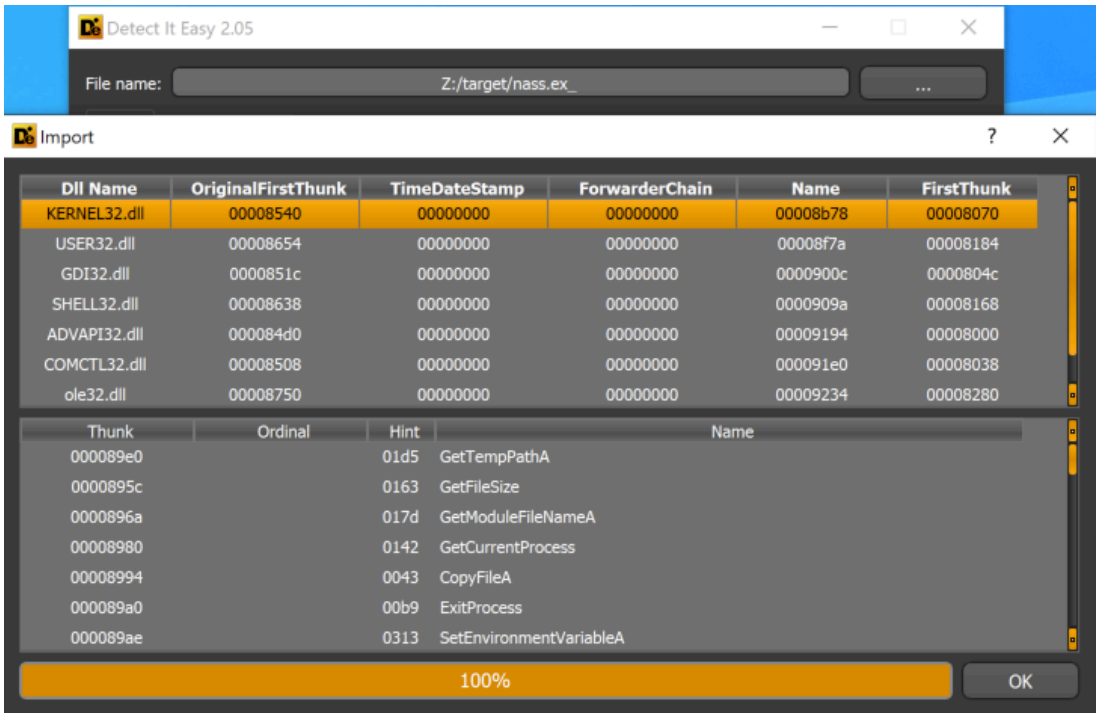

Figure 3. Imported functions from a Lokibot sample as seen in the import directory

From the *KERNEL32.DLL* library, this sample imports *GetTempPathA()*, *GetFileSize()*, *GetModuleFileNameA*, and other functions. The imported functions from all of the imported libraries are considered in generating the ImpHash. This way, similar files, regardless if new data is added, would

have the same ImpHash value — unless the developers changed its features by using (and therefore importing) a new function or removing a previously used one.

The other hashing technique I wish to highlight is impfuzzy, which generates a similarity digest based on the same input ImpHash is based on. It was created in 2016 but did not gain much popularity. In fact, I was only recently made aware of its existence during the review process of this article.

Analyzing Linux IoT malware

The aforementioned techniques work well for Windows executables, and algorithms like TLSH, ssdeep, and sdhash actually work for any type of file.

In 2018, I was overwhelmed by the number of IoT malware families and the number of samples that had been compiled for different architectures. People who closely follow the IoT malware trend would be familiar with file listings such as the one shown in Figure 4, which is that of an IoT botnet.

Index of /bins

Name	Last modified	Size	Description
Parent Directory		-	
a.arm	10-Nov-2019 11:20	37K	
a.arm5	10-Nov-2019 11:20	29K	
a.arm7	10-Nov-2019 11:20	106K	
a.i686	10-Nov-2019 11:20	37K	
a.mpsl	10-Nov-2019 11:20	50K	
a.x86	10-Nov-2019 11:20	33K	
shibui.arc	10-Nov-2019 11:19	41K	
shibui.arm	10-Nov-2019 11:20	45K	
shibui.arm5	10-Nov-2019 11:20	37K	
shibui.arm6	10-Nov-2019 11:20	55K	
shibui.arm7	10-Nov-2019 11:20	115K	
shibui.i686	10-Nov-2019 11:19	45K	
shibui.kill	10-Nov-2019 11:20	41K	
shibui.m68k	10-Nov-2019 11:20	46K	
shibui.mips	10-Nov-2019 11:20	60K	
shibui.mpsl	10-Nov-2019 11:20	62K	
shibui.ppc	10-Nov-2019 11:20	45K	
shibui.sh4	10-Nov-2019 11:20	41K	
shibui.spc	10-Nov-2019 11:20	49K	
shibui.x86	10-Nov-2019 11:19	41K	

Apache/2.2.15 (CentOS) Server at [redacted] Port 80

Figure 4. An example of an open directory in an IoT botnet download server

Usually, IoT botnet samples are based on the Tsunami, Gafgyt, or Mirai botnets. It is not unheard of to find combinations of these variants to create a spectrum of IoT malware types. Some of them are sophisticated, while others are not so. Some become improved versions of their predecessors, while others become buggy at best. Keeping updated on all of the variants that are constantly being developed,

correctly detecting malware samples, and accurately grouping them according to their real malware family (e.g., correctly detecting a sample according to what specific Mirai variant it is as opposed to tagging it as generic “Mirai”) can be challenging — especially since IoT malware is mostly made using ELF files.

Thus, I thought of creating “ImpHash for IoT malware,” which basically uses ImpHash techniques in analyzing ELF executables. After looking into the specifics of the study and conducting my own research, I found that no such project existed. So I decided to start my own.

My goal was to get imported functions of an ELF file and use them to feed a similarity digest algorithm that I could use to cluster similar files. The symbol extraction becomes straightforward when the ELF file has a symbol table, as seen in the example in Figure 5.

```
$ file hdump_32_dyn
hdump_32_dyn: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 2.6.32
, BuildID[sha1]=31a8c3c64031ff57a275201c39ba7220ab3e7e83, not stripped
$
$ nm --dynamic hdump_32_dyn
08048f8c R _IO_stdin_used
         U __ctype_b_loc
         w __gmon_start__
         U __libc_start_main
         U calloc
         U exit
         U fclose
         U fgetc
         U fileno
         U fopen64
         U fprintf
         U fread
         U free
         U fseek
         U ftell
         U getopt
         U optarg
         U printf
         U puts
         U qsort
         U rewind
         U select
         U stderr
         U strchr
         U strtoul
```

Figure 5. Getting the symbols, including imported functions, from a dynamically linked ELF file that did not have its symbols removed (stripped)

ELF symbols include not only the function names but also other symbols, so I wrote a basic piece of code to extract, sort (making my implementation resistant to symbol ordering changes), and generate a hash out of them.

It worked, but I was faced with two challenges. One was making a locality-sensitive hash that is resistant to small changes applied on files. The solution needs to be able to still track a malicious file that had two or three functions added to it, therefore making a new variant. The other was ensuring that the solution is able to accurately identify “stripped binaries,” or binaries whose imports have been deliberately removed

by a malicious creator. But since stripped binaries have no symbols, how could I calculate a checksum out of them?

The telfhash approach

I aptly (albeit admittedly a tad unimaginatively) named our tool telfhash — for Trend Micro ELF Hash — because it specifically deals with ELF files. For the following cases, we have learned the functions used by the binary from its symbol table:

- Statically linked with symbols (non-stripped binaries)
- Dynamically linked with symbols (non-stripped binaries)
- Dynamically linked with no symbols (stripped binaries)

There is a caveat in the last case. Although we can strip a dynamically linked binary to remove its symbols, the symbols related to the external functions used are *not removed* from the binary — otherwise the loader would have no way to resolve it during loading time.

I decided to use TLSH instead of MD5 (the algorithm that ImpHash uses) or any other cryptographic hash. This way, I would be able to take advantage of its locality-sensitive nature without losing the structural approach of using a function list as an input for the algorithm.

With this, I was able to calculate the same telfhash value for different versions of a given ELF executable using the *-g / --group* switch, as exemplified in Figure 6.

```
$ telfhash -g hdump_32_dyn*
hdump_32_dyn           4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_32_dyn_stripped  4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f

Group 1:
    hdump_32_dyn
    hdump_32_dyn_stripped
```

Figure 6. Clustering two versions of the same program wih the *-g / --group* switch of the telfhash tool

This was a good start, but it was simply not enough — especially since IoT malware samples are usually compiled for different architectures, too.

Multi-architectural challenge

To address the dilemma of IoT malware being compiled for different architectures, telfhash needs to produce an output of similar values regardless of the architecture the binaries were compiled in (in case they were compiled using the same source code). With this addressed, telfhash could be used to find the same malware compiled for a different architecture.

In order to achieve this, I carefully inspected many malware files and some regular programs that I had written in C and compiled for different architectures like x86, x86-64, Advanced RISC Machine (ARM), and Microprocessor without Interlocked Pipelined Stages (MIPS). I prioritized them based on their popularity among malware samples.

By ignoring the architecture-specific functions added by the compiler — I used the GNU C Compiler (GCC) toolchain in my tests — I had the interesting results shown in Figure 7.

```
$ telfhash -g hdump_*_dyn*
hdump_32_dyn               4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_32_dyn_stripped      4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_32_so_dyn            4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_32_so_dyn_stripped   4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_64_dyn               4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_64_dyn_stripped      4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_64_so_dyn            4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_64_so_dyn_stripped   4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_arm32_dyn            4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_arm32_dyn_stripped   4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f

Group 1:
    hdump_32_dyn
    hdump_32_dyn_stripped
    hdump_32_so_dyn
    hdump_32_so_dyn_stripped
    hdump_64_dyn
    hdump_64_dyn_stripped
    hdump_64_so_dyn
    hdump_64_so_dyn_stripped
    hdump_arm32_dyn
    hdump_arm32_dyn_stripped
```

Figure 7. Clustering multi-architecture versions of a program using telfhash with the *-g* switch

Using telfhash, I was able to cluster different binaries compiled for different architectures. But despite moving toward the right direction, it was not enough. I still had not been able to analyze statically compiled binaries without symbols using telfhash.

How telfhash deals with statically linked stripped binaries

No symbols, no win — and I had to think of a creative solution for my statically compiled binary issue. Thankfully, during a nice brainstorming session with my colleague Joey Costoya, I was able to think of a possible idea: to get the destination addresses for the function calls in the binary, create a list out of them, and feed it to telfhash.

To get this to work, I made telfhash look for the argument of the instruction that calls a function in each supported architecture. Figure 8 shows a code excerpt that illustrates just that.

```
    if code_section is not None:
        for i in md.disasm(code_section.data(), ofs):
            if arch in ("x86", "x64") and i.mnemonic == "call":
                # Consider only call to absolute addresses
                if i.op_str.startswith('0x'):
                    address = i.op_str[2:] # cut off '0x' prefix
                    if not address in symbols_list:
                        symbols_list.append(address)

            elif arch == "ARM" and i.mnemonic.startswith("bl"):
                if i.op_str.startswith('#0x'):
                    address = i.op_str[3:]
                    if not address in symbols_list:
                        symbols_list.append(address)

            elif arch == "MIPS" and i.mnemonic == "lw":
                if i.op_str.startswith("$t9, "):
                    address = i.op_str[8:-5]
                    if not address in symbols_list:
                        symbols_list.append(address)

    return symbols_list
```

Figure 8. Source code that implements the “call counting” feature

We see this feature at work in Figure 9.

```
$ telfhash -g *
hdump_32_dyn              4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_32_dyn_stripped     4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_32_so_dyn           4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_32_so_dyn_stripped  4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_32_stat             7bc08cc11d4f280a4c63c9e4bc452bc31ee70c4a69bd3d410f80cc00a92ca4e364fc0e
hdump_32_stat_stripped    5891d4636d799ce8b7f05801825a31748a3ae03b69d039b15df364a0f7b3e03563ad79
hdump_64_dyn              4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_64_dyn_stripped     4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_64_so_dyn           4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_64_so_dyn_stripped  4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_64_stat             7bc08cc11d4f280a4c63c9e4bc452bc31ee70c4a69bd3d410f80cc00a92ca4e364fc0e
hdump_64_stat_stripped    d5717bb108fa24a466cbd511b322b4f5a93519e922ed35a1673778c4dfc5fc128b6823
hdump_arm32_dyn           4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_arm32_dyn_stripped  4cb01247570b11c8557a446148813f9610836401fcbc2b000c08c040000c183658e84f
hdump_arm32_stat          7bc08cc11d4f280a4c63c9e4bc452bc31ee70c4a69bd3d410f80cc00a92ca4e364fc0e
hdump_arm32_stat_stripped fa01bd51ef19079c66d1939146ce65788afe30aca700dbb28b587b5b5553ec0b21d833

Group 1:
    hdump_32_stat
    hdump_64_stat
    hdump_arm32_stat
Group 2:
    hdump_32_dyn
    hdump_32_dyn_stripped
    hdump_32_so_dyn
    hdump_32_so_dyn_stripped
    hdump_64_dyn
    hdump_64_dyn_stripped
    hdump_64_so_dyn
    hdump_64_so_dyn_stripped
    hdump_arm32_dyn
    hdump_arm32_dyn_stripped
Cannot be grouped:
    hdump_arm32_stat_stripped
    hdump_32_stat_stripped
    hdump_64_stat_stripped
```

Figure 9. Telfhash dealing with statically linked stripped ELF binaries

Finally, I have an algorithm that is architecture-agnostic. The only obvious downside of it is that a telfhash value of a statically linked stripped binary does not match the others (as shown in Groups 1 and 2 of Figure 9), but will still be grouped together with other files compiled for multiple architectures.

Using telfhash on real malware

All these efforts would have been in vain if telfhash did not work with real malware. Thus, we used telfhash to try to analyze some malware samples, such as the XorDDoS backdoor, as shown in Figure 10.

```
$ telfhash -g *
XorDDos.x86.2014.0.bin  2e3162e118bc0c860de0ac145c7c3b82ca8b91771fa4961caf99cd89714f125f67bc06
XorDDos.x86.2014.1.bin  2e3162e118bc0c860de0ac145c7c3b82ca8b91771fa4961caf99cd89714f125f67bc06
XorDDos.x86.2014.2.bin  2e3162e118bc0c860de0ac145c7c3b82ca8b91771fa4961caf99cd89714f125f67bc06
XorDDos.x86.2014.3.bin  2e3162e118bc0c860de0ac145c7c3b82ca8b91771fa4961caf99cd89714f125f67bc06
XorDDos.x86.2014.4.bin  2e3162e118bc0c860de0ac145c7c3b82ca8b91771fa4961caf99cd89714f125f67bc06
XorDDos.x86.2015.0.bin  033161e118bc0c860ee0ac104c7d3f82ca8b917b1fa8961daf99cd89714f111f67bc0a
XorDDos.x86.2015.1.bin  b73140e518bc0c860ee0ac145c7d3b82ca8b927b1fa8962c9f99cd89754f115f66bc16

Group 1:
    XorDDos.x86.2014.0.bin
    XorDDos.x86.2014.1.bin
    XorDDos.x86.2014.2.bin
    XorDDos.x86.2014.3.bin
    XorDDos.x86.2014.4.bin
    XorDDos.x86.2015.0.bin
    XorDDos.x86.2015.1.bin
```

Figure 10. Telfhash clustering all XorDDoS malware samples in one group

I also collected samples of Momentum, an IoT botnet that affected devices running on Linux, and ran telfhash against them, as shown in Figure 11.

```
$ file *
004c3d8ad74558d9560bfd23343c5d5a7637761855712b9e4f8cc895eeaf48a0: ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), statically linked, not stripped
07cebb563a245f52d7d18c17167661a4d2c010424b7ebd5569315f116b6a10bc: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked, not stripped
086f79f4ad0acfe6f23b1298d3ee858f56eabbc39c738858c6c0932aa7d29f61: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, with debug_info, not stripped
0afffeda51d6c11572e077720a2e3b84c4bc7d5e2fa171ea4c2e0fb993466be6: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, with debug_info, not stripped
0b49efe340e81ba3a5e1cc9013e550e3494fe1d4490db1617b66d9130f33138c: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
10e95641e1d03950b09d4bd867ff816ce089c92f6e341cacfac8344b901a1dcd: ELF 64-bit MSB executable, MIPS, MIPS64 version 1 (SYSV), statically linked, with debug_info, not stripped
13490afa959a0d7636d67fd01ea469714b77b5439c9eee58d9552175da0831d2: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked, not stripped
152e6de8fbefee3a78c79724475b58fecec68f750e5418dc6f55ee74667b8fa2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
1dfacefef9276295fb344a56854db423359bd4bcec68dfba27ac88d7b410114c: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked, no section header
1e9901479b1f90d42ea45ffb38782dd0879a289aa574567c2c032Se85f213beb: ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, with debug_info, not stripped
1fc4b2d4e1935fdda4f908785ce761c6dff137785855a9ad15d865c0447be1702: ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), statically linked, not stripped
229aeaee221dab8d70e0ab874ded4a067e36eea8fad31089a49b9e84e3d0015c: ELF 32-bit LSB executable, Renesas SH, version 1 (SYSV), statically linked, not stripped
22b594776e894c544fd7e5cac3f89528e8600dfc1f00a212af62037bb9ff9186: ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, with debug_info, not stripped
26c9946dfe325365d6de4cbb977fa28f447586ffc59d6b3cfedbd13cc16a0ad5: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
309a610113d44c2188b244681e4eaa42a1530496454450089d35f8948dbdc0bf: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
3c6d31b289c46b988be7908acd84086653a0774206b3310e0ea4e6779e1ff4124: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
514988eb51eaacc548eb9616f96ac45445670lc338a93280454da48b2de28f14: ELF 32-bit LSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked, not stripped
5595396bbb4d51ec52a55462ba86cd00bd706ee6a1c654adbbf67ae6e1c2c68c4: ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, with debug_info, not stripped
560125050b24089bf651c419b35c00bdfe11e042178574c88b39dbc4e7c9cef: ELF 64-bit MSB executable, MIPS, MIPS64 version 1 (SYSV), statically linked, with debug_info, not stripped
5b5492a045bda33263e5c95e9de642d602c46eaf79b8cff2bdfca2f901c8caac: ELF 32-bit MSB executable, Motorola m68k, 68020, version 1 (SYSV), statically linked, not stripped
69c0be026ea069c8b0025a953bc1cb8dbaeaa50d21cf29fb10b58814bf499ac0: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, with debug_info, not stripped
74de649bcfbefb0a4806225ccd3bf3941d5018d23351dd2578d0880cf52b55dd: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
7ffd92541ce11db6b472256d5b4fa1b0f114beee196dc54607904b194e70fc5d: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux), statically linked, no section header
81b490013553014dc9b88d6b3544a588430f03f5fede d07efc97227fa6edec25: ELF 32-bit MSB executable, Motorola m68k, 68020, version 1 (SYSV), statically linked, not stripped
8cd0795b0652c23b339a5e80ba77689a8d44d191e8403c1579b3d00daf1c58f5: ELF 32-bit LSB executable, Renesas SH, version 1 (SYSV), statically linked, not stripped
906dcaee54ce4c9cb3e237f0b291631378609db91a4b0ccd615f495eacd8bbc0: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, with debug_info, not stripped
ae4f0e47e6cebb83dcba471042bedf977326b0ed2993d427f80d257d6641096d: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, with debug_info, not stripped
b374f0201adea5518a8319a585a0268170d213305412e1645b7088fe6ddca3bb: ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), statically linked, not stripped
ba51a1f9e48a3b60111ee1c64134584af4c032b075f40f72eca169253f9b54d6: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked, not stripped
c260f450223761212ada9d8bd3fda9d8958c342862830f490a113b4328b792ec: ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), statically linked, not stripped
c8386835395b9f7209361ec9e48c1bd734c139fd5d092053759986870061ba457: ELF 32-bit MSB executable, PowerPC or cisco 4500, version 1 (SYSV), statically linked, not stripped
d6df3a1cb7be08348c61757568c48da2b241cc250d071fc11e480564125f465e: ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, with debug_info, not stripped
da85e99d16d534e22422e64af3d52a57442e4db5486020bba47322da4cdbdbdd: ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, with debug_info, not stripped
deebfd7e35f84ce4791829150d473ce54b66ffb282ce0330e12210797fbb08b7: ELF 32-bit LSB executable, ARM, version 1 (ARM), statically linked, no section header
e3f03f0713076e5a9090703861d69663e34eff54ed3a7578036f01c9488caada: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, with debug_info, not stripped
e48e3f4ea3168e2fb3e9e93fc4d3a146e1953a028e051e6cb6d616a11019d290: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, with debug_info, not stripped
eb509ce80872ba42e04aecec089c169a32f201cd0f79c81557932c95da97916c: ELF 32-bit LSB executable, ARM, EABI4 version 1 (SYSV), statically linked, with debug_info, not stripped
f082ada6df2d85d1fad70dc3d7781d7b85ed4c70442ef1842dcef683e516f888: ELF 32-bit MSB executable, Motorola m68k, 68020, version 1 (SYSV), statically linked, not stripped
f572c0780d42e9597c34e0875492a7991982873008586 3e5b30abc3340ebfbc1: ELF 32-bit MSB executable, MIPS, MIPS-I version 1 (SYSV), statically linked, no section header
fbec31f005ec81c5c6283a87902abc46ab9ae371c053f6b243a94045333f9d71: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, not stripped
```

Figure 11. Momentum botnet samples compiled for different architectures

By using telfhash and the TLSH distance measure (with a threshold set to 50), I ended up clustering the Momentum botnet samples in three similar groups, as indicated in Figure 12.

```
Group 1:
    086f79f4ad0acfe6f23b1298d3ee858f56eabbc39c738858c6c0932aa7d29f61
    309a610113d44c2188b244681e4eaa42a1530496454450089d35f8948dbdc0bf
    560125050b24089bf651c419b35c00bdfe11a44217857f4c88b39dbc4e7c9cef
    e3f03f0713076e5a9090703861d69663e34eff54ed3a7578036f01c9488caada
    e48e3f4ea3168e2fb3e9e93fc4d3a146e1953a028e051e6cb6d616a11019d290
Group 2:
    0afffeda51d6c11572e077720a2e3b84c4bc7d5e2fa171ea4c2e0fb993466be6
    10e95641e1d03950b09d4bd867ff816ce089a92f6e341cacfac8344b901a1dcd
    1e9901479b1f90d42ea45ffb38782dd0879a289aa574567c2c0325e85f213beb
    69c0be026ea069c8b0025a953bc1cb8dbaeaa50d21cf29fb10b58814bf499ac0
    74de649bcfbefb0a4806225ccd3bf3941d5018d23351dd2578d0880cf52b55dd
    906dcaee54ce4c9cb3e237f0b291631378609db91a4b0ccd615f495eacd8bbc0
    ae4f0e47e6cebb83dcba471042bedf977326b0ed2993d427f80d257d6641096d
    eb509ce80872ba42e04aecec089c169a32f201cd0f79c81557932c95da97916c
Group 3:
    004c3d8ad74558d9560bfd23343c5d5a7637761855712b9e4f8cc895eeaf48a0
    07cebb563a245f52d7d18c17167661a4d2c010424b7ebd5569315f116b6a10bc
    0b49efe340e81ba3a5e1cc9013e550e3494fe1d4490db1617b66d9130f33138c
    13490afa959a0d7636d67fd01ea469714b77b5439c9eee58d9552175da0831d2
    152e6de8fbefee3a78c79724475b58fecec68f750e5418dc6f55ee74667b8fa2
    1fc4b2d4e1935fdda4f908785ce761c6dff13778585a9ad15d865c0447be1702
    229aeaee221dab8d70e0ab874ded4a067e36eea8fad31089a49b9e84e3d0015c
    22b594776e894c544fd7e5cac3f89528e8600dfc1f00a212af62037bb9ff9186
    26c9946dfe325365d6de4cbb977fa28f447586ffc59d6b3cfedbd13cc16a0ad5
    3c6d31b289c46b98be7908acd84086653a0774206b3310e0ea4e6779e1ff4124
    514988eb51eaacc548eb9616f96ac454456701c338a93280454da48b2de28f14
    5595396bba4d51ec52a55462ba86c00bd706ee6a1c654adbbf67ae6e1c2c68c4
    5b5492a045bda33263e5c95e9de642d602c46eaf79b8cff2bdfca2f901c8caac
    81b490013553014dc9b88d6b3544a588430f03f5feded07efc97227fa6edec25
    8cd0795b0652c23b339a5e80ba77689a8d44d191e8403c1579b3d00daf1c58f5
    b374f0201adea5518a8319a585a0268170d213305412e1645b7088fe6ddca3bb
    ba51a1f9e48a3b60111ee1c64134584af4c032b075f40f72eca169253f9b54d6
    c260f450223761212ada9d8bd3fda9d8958c342862830f490a113b4328b792ec
    c8386835395b9f7209361ec9e48c1bd734c139fd5d09205375998687061ba457
    d6df3a1cb7be08348c61757568c48da2b241cc250d071fc11e480564125f465e
    da85e99d16d534e22422e64af3d52a57442e4db5486020bba47322da4cdbdbdd
    f082ada6df2d85d1fad70dc3d7781d7b85ed4c70442ef1842dcef683e516f888
    fbec31f005ec81c5c6283a87902abc46ab9ae371c053f6b243a94045333f9d71
```

Figure 12. Clustering Momentum botnet samples in three groups (telfhash values redacted for brevity)

Currently, telfhash supports x86, x86-64, ARM, and MIPS. I decided to primarily build support for these architectures because they cover the majority of IoT malware samples. The binaries compiled for PowerPC and Renesas are therefore not yet on the list.

Making telfhash available for all

As an open source supporter, I always intended to make telfhash an open-source tool. Thankfully, I had all the support that I needed from Trend Micro to make that a reality.

We have been using telfhash internally. But there is no reason not to share it with the cybersecurity community so that we can discuss new features, make improvements, fix bugs, and take advantage of all of the benefits that only the open source community can offer. Therefore, we have publicly released telfhash and made its source code available on Github.

We are offering telfhash as a Python library (a huge thank-you goes out to Joey for turning my ideas into professional Python code and helping improve it) so that it can be easily integrated in Python scripts in order to generate a similarity digest for ELF files.

I would be happy to discuss any weaknesses, bugs, improvements, and ideas pertaining to telfhash. It is my sincere hope that telfhash proves to be an essential tool in combating Linux IoT malware.