BinclustRE: a configurable tool for binary clustering

Alex Marcelino Santee[1], Fernando Antonio Dantas Júnior[1], Françoa Taffarel,[1] Osmany Barros de Freitas[1] e Lourenço Alves Pereira Júnior[1]

[1]Divisão de Ciência da Computação – ITA – São Jose dos Campos, SP – Brazil

{alex.santee.101394,fernando.junior.101049}@ga.ita.br,
{taffarel,osmany,ljr}@ita.br

Abstract. *This project presents* `BinclustRE`*. This Open-Source tool creates a pipeline for automated clustering of executable binaries, which is helpful for security because programs in the same group should have similar security vulnerabilities. It implements a pipeline in which it's simple to integrate new techniques and run them using multi-threading and caching of intermediary results. Our experiments show* `BinclustRE` *was able to cluster versions of OpenSSL's* `libssl` *vulnerable to heartbleed in different groups from versions at least one year newer than the patch.*

Resumo. *Esse projeto apresenta o* `BinclustRE`*, uma ferramenta de Código Aberto que cria um pipeline para agrupamento automatizado em binários de executáveis, o que é útil para a segurança porque programas agrupados juntos devem apresentar vulnerabilidades de segurança parecidas. Ele implementa um pipeline no qual é simples integrar novas técnicas e executa-as com multi-threading e cache dos resultados intermediários. Nossos experimentos mostram que o* `BinclustRE` *foi capaz de agrupar versões da* `libssl` *do OpenSSL vulneráveis ao Heartbleed em grupos diferentes a versões pelo menos um ano mais novas que a correção.*

1. Introduction

In the context of the Internet of Things (IoT), the number of devices is growing, with the presence of 35.82 billion IoT devices since 2021, and an estimate of this number getting to 75.44 billion by 2025 [Yaacoub et al. 2023]. According to [Liu et al. 2020], the IoT devices that develop similar functions have the same software, and most use the `Linux` Operating System. The possibility of utilizing open-source Software in IoT devices can make this software present in machines from different manufacturers. This practice has become increasingly common because using open-source code brings cost reduction, flexibility, and transparency advantages. However, with the increase of cybernetic attacks on IoT devices, especially Distributed Denial of Service (DDoS) attacks [Kumari and Jain 2023], this open-source software reuse implies the growth of the large-scale exploration by malicious agents since a vulnerability present in one device may be present in similar devices.

That said, a strategy to find vulnerabilities on a large scale is to extract binaries from the firmware of many devices and to determine similar groups. These clusters may be helpful for vulnerability detection in two ways: if a vulnerability is discovered in a binary, other binaries in the same group have a higher chance of having it, too, and significant clusters indicate common versions of a binary, which are an interesting

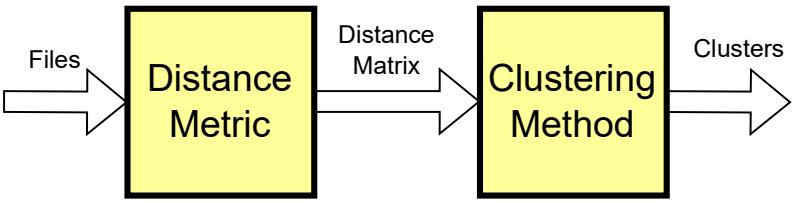


Figure 1. Architecture diagram of `BinclustRE`

starting point to find vulnerabilities at scale. However, detecting similar software is challenging because firmware analysis is usually a black box. After all, the exact source code is, in principle, unknown since only compiled binaries are available. This binary comparison is more complex than a simple byte-by-byte analysis because the same source code can create different binaries because of other compilation characteristics, like target architecture, compiler version, and optimization level.

This challenge of determining similar programs is called Binary Code Similarity Analysis (BCSA), and it's also useful for malware detection, plagiarism detection, and authorship identification [Kim et al. 2023]. The similarity analysis can be seen as a first step for clustering. However, to our knowledge, there is no initiative to make clusters for vulnerability detection, even though there are many clustering studies for malware analysis, as [Gibert et al. 2020] show in their literature review of machine learning for malware detection. Some malware techniques use dynamic analysis to group malware by its behavior, like HTTP headers, API calls, and syscalls, and others use static analysis techniques, like Control-Flow Graph comparison, to determine groups of malware executables.

Thus, this project presents `BinclustRE`, an Open-Source tool that allows the use of different binary analysis strategies to determine the detection of IoT device firmware binary clusters, aiming to help identify software binary vulnerabilities. Our tool seeks to address these challenges by providing a robust solution for binary code similarity analysis and clustering, thus improving the efficiency and effectiveness of detection and subsequent mitigation of vulnerabilities in IoT devices with its results. Our tool also proposes using the `Damicore` [Sanches et al. 2011] methodology, which uses the normalized compression distance (NCD) [Cilibrasi and Vitanyi 2005] as a similarity metric and makes clusters through algorithms of neighbor-joining and community detection, a clustering strategy based on phylogenetics and complex networks. We add to the `Damicore` methodology by creating a preprocessing step that allows it to work well with compiled binary files instead of source code.

2. Architecture and Functionalities

The tool architecture is a two-step pipeline: the first step is using a similarity metric on every pair to create a distance matrix, and the second one is making a clustering based on this distance, as illustrated in Figure 1. As an example of its versatility, one can embed `Damicore`'s pipeline into this architecture, with the NCD as the similarity metric and neighbor-joining and community detection as the clustering strategy. In practice, we observed that applying NCD to raw binaries creates groups primarily based on the CPU's architecture, so our pipeline adds a preprocessing step to the original to avoid this effect. This modifications to `Damicore`'s pipeline is shown in Figure 2.

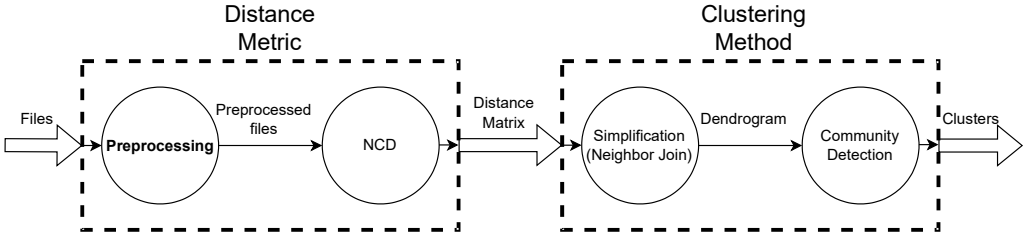


Figure 2. Example of embedding the `Damicore` pipeline to our architecture with the inclusion of our novel preprocessing step

There are many variations to experiment with in the pipeline. The theory of NCD states that all real-world compressors are adequate for similarity calculation [Cilibrasi and Vitanyi 2005], so there's more to explore than just the default compressors (i.e., *ppmd*, *gzip* and *bzip2*). From the NCD's distance matrix, the next step creates via neighbor-joining a dendrogram tree structure in which different community detection algorithms can be used being *fast-newmann* , *betweenness* and *random-walk* available by default. Finally, the preprocessing stage our tool adds is a variation to the method that can be experimented with. Our implementation uses some readily available tools for this job. Section 2.1 describes what we added in detail.

Our tool also has some functionalities to make it more practical to use. For better performance, the distance calculation and preprocessing are done in a multi-threaded way, keeping caches of its results so that the duplicate files don't have to be processed twice in case of multiple executions. Also, there is a Docker script to manage the dependency installation since it's intended to be used with many applications.

2.1. Preprocessing

A preprocessing stage is essential for `Damicore`'s pipeline because it's based on the NCD, which threatens the executable file as a sequence of bytes and tries to find patterns between them. This has effects like the architecture being the most critical factor in the similarity analysis because two binaries in the same architecture have the same opcodes for their instructions. Still, similar instructions in different architectures have completely different opcodes. Therefore, the following strategies are implemented in `BinclustRE`:

- String extraction: This should mostly get the texts that the program outputs or sends through the network, which is compilation-independent and takes little computational time.
- Control-Flow Graph (CFG): Analyzes program flow as a graph of possible paths with little dependence on typical compilation options.
- Binary Lifting: Tries to bring the executable to a previous compilation state, like intermediary representation or source code.

For each execution, the user can choose one of these preprocessing strategies, which `BinclustRE` uses readily available applications for these jobs and is coded to simplify adding new ones. The string extraction is made via `strings`, a common Unix* Operating Systems tool. The CFG is obtained through `radare2`[1] or `angr`[2]. There are

[1] `https://rada.re/n/radare2.html`
[2] `https://angr.io/`

```
void thunk_FUN_001010a0(void) {
  FUN_001010a0();
  return;
}
```

(a) Raw output

```
void thunk_FUN_00000005(void) {
  FUN_00000005();
  return;
}
```

(b) After normalization

Figure 3. Comparison of Ghidra's raw output and its normalization

```
docker build -t binclustre ./
```

```
docker run -v <input-folder>:/usr/src/app/input-binaries/ \
-v \$PWD/caches:/usr/src/app/caches/ -v $PWD/results:/usr/src/app/results/ \
binclustre input-binaries/ <args>
```

Code 1. Commands for building and running the docker image

two different binary lifting strategies: LLVM intermediary representation with `retdec`[3] and C Pseudocode with `ghidra`[4].

2.1.1. Normalization

One issue from the preprocessing stage is that the applications intended to create a representation independent of compilation options often leak dependent information. For example, `ghidra` uses the function's address to make its name when unknown, but this value depends on the architecture's usual address space. It can affect the similarity score from compression. So `BinclustRE` implements a step of normalization in which these problematic values are changed to sequential addresses. Figure 3 shows an example of this normalization process for `ghidra`.

3. Demonstration

3.1. Requisites

The Open-Source tool is available in a GitHub repository called `BinclustRE`[5] that includes a full manual in its README file. For a simple execution, the program can be built and run with a `docker` script that handles all the dependency installing and can be executed using the commands in Code 1. For the visualization of dendrogram intermediary results, it's recommended to install a newick file viewer like FigTree[6].

3.2. Usage

For this example, the tool is run using CFG from `radare2` preprocessing method at a small collection of binaries from the `BinKit` [Kim et al. 2023] dataset, the binaries `cat`, `echo` and `cp` with some variations of architecture and optimization level. Code 2 shows the preprocessing step for one file.

The final result of `BinclustRE` is grouping the files in clusters, as shown in Figure 4(a). The first column is the input filename, and the second is the group index.

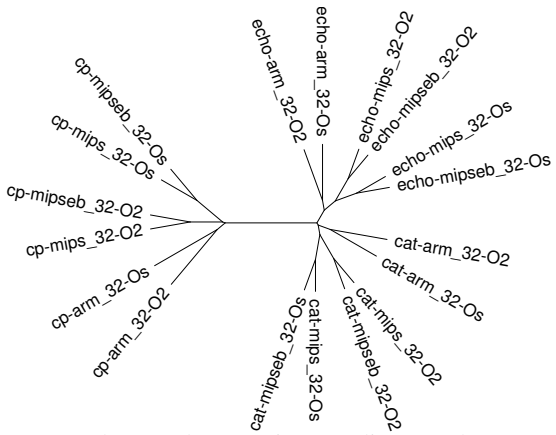
[3] `https://github.com/avast/retdec`
[4] `https://ghidra-sre.org/`
[5] `https://github.com/c2dc/BinclustRE`
[6] `https://github.com/rambaut/figtree`

```
$ docker run \
-v $PWD/input-binaries/:/usr/src/app/input-binaries/ \
-v $PWD/caches: /usr/src/app/caches/ \
-v $PWD/results:/usr/src/app/results/ \
binclustre input-binaries/
[+] Starting static analysis with radare-cfg method
[+] Analysing cat-mipseb_32-02
cat-mipseb_32-02: INFO: Analyze all flags starting with sym.
[...]
[+] cat-mipseb_32-02 analysed in 4.85 s
[+] Normalizing cat-mipseb_32-02
[+] Found addresses to replace: 4
[+] cat-mipseb_32-02 normalized in 0.00 s
[+] Analysing echo-arm_32-02
echo-arm_32-02: INFO: Analyze all flags starting with sym
 and entry (aa)
```

Code 2. Example of the tool's typical usage

```
filename,cluster
cat-arm_32-O2,0
cat-arm_32-Os,0
cat-mips_32-O2,1
cat-mips_32-Os,0
cat-mipseb_32-O2,1
cat-mipseb_32-Os,0
cp-arm_32-O2,2
cp-arm_32-Os,2
cp-mips_32-O2,2
cp-mips_32-Os,2
cp-mipseb_32-O2,2
cp-mipseb_32-Os,2
echo-arm_32-O2,3
echo-arm_32-Os,3
echo-mips_32-O2,3
echo-mips_32-Os,3
echo-mipseb_32-O2,3
echo-mipseb_32-Os,3
```

(a) Clustering result



(b) Dendrogram intermediate result

Figure 4. Example usage for cat echo and cp binaries

The Machine Learning technique can't determine what the groups mean to create the partitions, so the clusters have no name. This result shows the default clustering settings bring promising results because, other than ARM binaries being kept separate from MIPS, the clusters correctly match the programs `cat` `echo` and `cp`.

To evaluate the quality of this clustering and the CFG preprocessing, we compare the clustering result with the ground truth in Table 1. Each row represents the real group, each column represents the groups from the tools, and the cells represents the number of binaries in both groups. It is observed that the clustering result with no preprocessing groups is mainly based on the processor's architecture. The adjusted-rand index of the clustering using the CFG preprocessing is 0.867, and its value is 0.386 when no preprocessing is applied.

Besides the clustering result, using the intermediary dendrogram for visual inspection is interesting to understand how the groups are made. A dendrogram is an unrooted ternary tree whose leaves represent the input binaries, and the other nodes are what would be "common ancestors" from an evolutionary tree. Figure 4(b) shows the view of this intermediary result in FigTree. Visually, three program types form groups, but ARM versions are slightly apart from MIPS.

Table 1. Comparison of contingency matrixes

(a) CFG preprocessing

real\classified	0	1	2	3
cat	4	2	0	0
cp	0	0	6	0
echo	0	0	0	6

(b) No preprocessing

real\classified	0	1	2	3
cat	2	2	2	0
cp	0	0	0	6
echo	2	2	2	0

4. Results

To evaluate quality clustering, we created some datasets to check if the processing time is reasonable and if the clustering corresponds to the semantic meaning we seek. The quality of these clusterings in which the ground truth is known is a starting point to estimate the clustering quality in a real-world scenario of Firmware analysis in the original source code and compiler version, and options are unknown.

For IoT binaries, the main challenge is that executables are compiled for different processor architectures, usually ARM or MIPS, and it's common to optimize for smaller binary size when the amount of memory is limited. Thus, our datasets try to mix up binaries with different compiler options, and the tool aims to group together binaries that were built from the same or similar source codes.

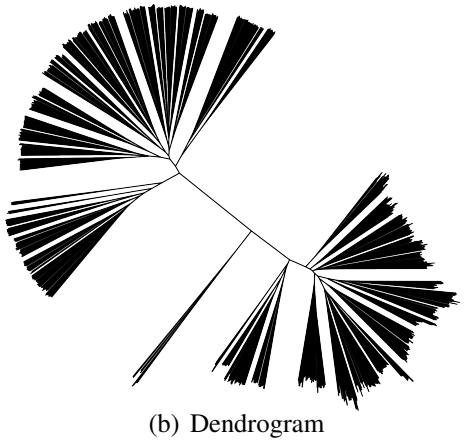
4.1. Large dataset

Our tool is intended to be used for a large number of binaries, so as a performance test, we created a dataset of 1728 binaries from Binkit, 864 different versions of the `bool` command line tool, and 864 for `grep`, each one with varying options of compiler and versions. The run time for this clustering was 1 hour in a 20-core computer with 64 GB of RAM using the CFG extraction with `radare2` and with the betweenness community detection algorithm. Since the origin of the binaries for this experiment is known, we can also evaluate quality clustering using the ground truth.

Figure 5(a) shows the contingency matrix of the clustering using two groups, and Figure 5(b) shows the dendrogram intermediary result. The division of the clustering ninead only nine wrong classifications, with an accuracy of 0.995 and an adjusted-rand index of 0.979, and the dendrogram shows that the binaries were separated into roughly two distinct groups.

real\classified	0	1
bool	863	1
grep	8	855

(a) Contingency matrix



(b) Dendrogram

Figure 5. Results for a large number of binaries for bool and grep

4.2. Heartbleed detection

In the security context, we will now analyze the Heartbleed vulnerability, a critical flaw revealed in 2014 in the OpenSSL cryptography library. Due to an overlooked flaw in the protocol's implementation, adversaries could exploit the server to read its memory, potentially acquiring sensitive data, including private keys used to encrypt communications. To check if our tool clusters different versions of the same binary in other groups, we created a small dataset with varying versions of OpenSSL's `libssl`, each version compiled for the ARM, MIPS, and x86 micro-architectures. The first version is from 2014, before the patch. The others are progressively older, one year, two years, and four years after the patch.

The preprocessing method used for this dataset was string extraction, and Figure 6(a) shows that the clusters discovered by the tool correspond to the accurate versions for each year. Figure 6(b) is the dendrogram, which shows that binaries from similar years show up closer in the tree. These results indicate that our technique could group different versions of binaries in time, which is helpful for vulnerability detection since older versions should have a higher chance of unpatched vulnerabilities. The binaries vulnerable to Heartbleed were in the same group in our experiment.

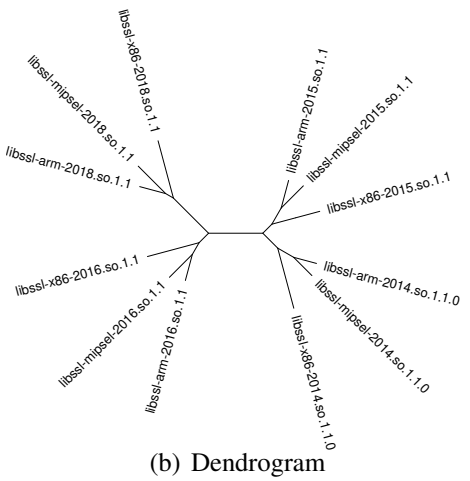
5. Conclusion

In this paper, we present `BinclustRE`, a versatile tool for binary clustering with cache and multi-threading. Its architecture is intended for experimentation of different techniques, being the code written so that new strategies can be easily integrated into the pipeline. This implementation shows that `Damicore`'s pipeline fits nicely in the architecture, and each pipeline stage from the architecture can be easily substituted by other techniques, like CFG edit distance as a distance metric instead of NCD or using DBSCAN as a clustering detection technique instead of neighbor join and Fast Newman.

This versatility and these performance features make it a viable tool to explore different strategies for black-box binary clustering, with the addition of newer techniques and a more detailed comparison of the current methods interesting future works. Another interesting future work would be the creation of a dataset that explores variations of program versions over time. The best dataset we know of is from [Kim et al. 2023],

real\classified	0	1	2	3
2014	0	3	0	0
2015	3	0	0	0
2016	0	0	0	3
2018	0	0	3	0

(a) Contingency matrix



(b) Dendrogram

Figure 6. Results for different OpenSSL versions

which proposed BinKit as a unified dataset for BCSA. It explores many compiler options and versions but, unfortunately, does not provide different versions of these compiled binaries, which is the most exciting feature for vulnerability detection.

The results of our initial experiments were promising to be able to process thousands of binaries in a couple of hours, and a simple dataset that explored versions over time was perfectly grouped for `libssl` of different years. In this clustering, all of the binaries vulnerable to Heartbleed were in the same group, showing that it was effective in together these vulnerable binaries.

Acknowledgments

This work has financial support from the Graduate Program in Operational Applications—PPGAO/ITA and from FAPESP process #2020/09850-0.

References

Cilibrasi, R. and Vitanyi, P. (2005). Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545.

Gibert, D., Mateu, C., and Planes, J. (2020). The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526.

Kim, D., Kim, E., Cha, S. K., Son, S., and Kim, Y. (2023). Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering*, 49(4):1661–1682.

Kumari, P. and Jain, A. K. (2023). A comprehensive study of ddos attacks over iot network and their countermeasures. *Computers & Security*, 127:103096.

Liu, K., Yang, M., Ling, Z., Yan, H., Zhang, Y., Fu, X., and Zhao, W. (2020). On manually reverse engineering communication protocols of linux-based iot systems. *IEEE Internet of Things Journal*, 8(8):6815–6827.

Sanches, A., Cardoso, J. M., and Delbem, A. C. (2011). Identifying merge-beneficial software kernels for hardware implementation. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pages 74–79.

Yaacoub, J.-P. A., Noura, H. N., Salman, O., and Chehab, A. (2023). Ethical hacking for iot: Security issues, challenges, solutions and recommendations. *Internet of Things and Cyber-Physical Systems*, 3:280–308.