Identifying Microcontroller Architecture Through Static Analysis of Firmware Binaries

David Malaschonok
Fraunhofer SIT — ATHENE
david.malaschonok@sit.fraunhofer.de

***Abstract*—With the advance of IoT technology, embedded systems have become omnipresent in everyday life, taking on ever more security sensitive tasks. Because of this, the security analysis of embedded firmware has reached unprecedented importance.**

At the same time, the need to keep production and operation costs low imposes strong resource constraints and optimization pressure on the design of embedded IoT devices. Trade-offs include smaller firmware images that lack debug symbols, and lighter housing that is harder to disassemble. Notably, the cheapest products tend to receive the least amount of vendor support, thus making them more vulnerable, while simultaneously being the least amenable to analysis, thus making it harder for third parties to assess and address the resulting risks.

Knowing which precise microcontroller unit (MCU) is built into a particular device allows insight into its memory map, which is valuable for both static and dynamic analysis of its firmware. However, while it is usually easy to determine the manufacturer and model of an IoT appliance through visual inspection, identifying the MCU at the core of the device is often only possible after destructive disassembly.

To address this problem, we propose an automatic approach to derive the MCU of an embedded device from its firmware image. The approach is based on identifying which addresses the firmware expects to be accessible and finding the most similar MCU memory map in a pre-calculated knowledge base. Our approach does not depend on debug symbols or physical access to any part of the embedded device.

In our evaluation, this approach correctly identifies the precise MCU series 57% of the time and finds the most precise available memory map 44% of the time.

I. Introduction

The Internet of Things spans devices of various shapes and functions. A large part of these devices are *embedded devices*, i.e., single-board devices featuring a CPU, volatile and non-volatile memory, as well as multiple peripheral devices. The board containing the CPU and its peripherals is referred to as the *microcontroller unit* (MCU). The software executed on an embedded device is referred to as its *firmware*. Embedded devices typically operate under strong resource constraints, which limits the complexity of firmware that they are capable of executing.

Workshop on Security and Privacy in Standardized IoT (SDIoTSec) 2026
23 February 2026, San Diego, CA, USA
ISBN 978-1-970672-01-5
https://dx.doi.org/10.14722/sdiotsec.2026.23007
www.ndss-symposium.org

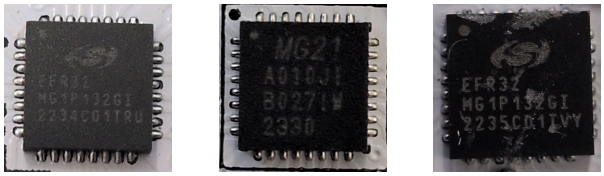

Fig. 1. MCU inscriptions found in disassembled embedded devices.

Fuzz testing firmware both on real [1]–[3] and on emulated hardware [4]–[6] has by now become an established approach. Both variants benefit from insights into the firmware's hardware-facing interactions, e.g., for harness generation or crash interpretation.

Embedded firmware typically interacts with peripherals through their memory-mapped interfaces, i.e., by performing read and write operations on specific addresses in the system's bus address space. Which register of which peripheral can be found under which address depends on the precise MCU built into the device. Hence, in order to fully understand (or emulate) the firmware's interactions with its hardware, it is essential to know the MCU's memory layout. In this paper, we will refer to the MCU on which the firmware is meant to be executed as the firmware's *target MCU*.

Usually the MCU built into a device can be identified by reading it off a chip on its main board (see fig. 1). However, in virtually all cases, reading the chip inscription requires disassembly beyond the point intended by the device manufacturer, potentially risking human injury or damage to the device. This can be particularly difficult for devices with hard to open housings, such as devices encased in a monolithic piece of plastic.

Further, physically accessing devices can be impractical in cases where firmware is being analyzed as a standalone product. If, for example, one where to attempt a fuzzing campaign on all firmware published on the website of a specific device vendor, it would be costly and difficult to acquire every single device for which said vendor is providing firmware. In such cases, a method to automatically detect the target MCU of a firmware binary without requiring any access to the hardware could help reduce cost and manual effort.

The goal of this paper is to determine whether the target MCU of a firmware image can be reliably identified without physical access to the device. To this end, we implement and

evaluate a fully automatic approach to match a firmware image to its target MCU using only static analysis and a knowledge base pre-calculated using public information.

Our approach only requires the firmware to be executable on its target architecture. In particular, it does not rely on the firmware to include metadata or debug symbols or for it to be built using any specific library.

The contributions of this paper are

- a novel method for identifying the target MCU of monolithic firmware with minimal manual effort,
- a pre-calculated machine-readable MCU knowledge base enabling our method including tools to rebuild the knowledge base from current sources, and
- an evaluation of the efficacy of our method on a ground truth of 42 firmware images.

The remainder of this paper is structured as follows. Section II introduces background concepts needed to follow this paper. Section III presents our approach in detail. Section IV evaluates the efficacy of our approach. Section V discusses the limitations of our approach as well as future work, and Section VI concludes this paper.

II. Background

In this section, we will introduce related work as well as all concepts necessary to follow the rest of this paper.

A. Machine-readable hardware descriptions

The most detailed and readily available source of memory layout information for virtually any MCU is its *reference manual*. A typical reference manual is a large PDF document, often exceeding 1000 pages, containing detailed information about every peripheral device built into the MCU, including addresses and semantic meaning of all registers exposed by these peripheral devices. Although PDF reference manuals are usually comprehensive and readily available on the websites of silicon vendors, they are not intended to be machine-readable.

The most common machine-readable hardware description format is *CMSIS System View Description (SVD)* [7], an XML-based file format containing peripheral specifications for use in, e.g., code generation. The standardization of this format has resulted in many silicon vendors creating and publishing SVD files containing their device specifications. The two largest public SVD repositories known to us are `cmsis-svd-data`[1] and Arm Keil[2].

`cmsis-svd-data` is an independent GitHub repository aiming to aggregate a comprehensive collection of all freely available SVD files created by silicon vendors and third parties alike.

Arm Keil is a microcontroller development kit containing, among other things, an extensive registry of references to CMSIS packs provided by silicon vendors, which in turn contain SVD files.

[1] https://github.com/cmsis-svd/cmsis-svd-data

[2] https://www.keil.arm.com/

B. Set similarity

Our approach makes use of a set similarity metric to identify which microprocessor architecture matches a given firmware image best.

A *set similarity* function (also *binary similarity*) is a function that defines a real-valued similarity score for any two sets. The higher the similarity score, the more similar the two sets are deemed to be.

Empirical evaluation of different similarity functions for use in software analysis and beyond has been an active topic of research for at least two decades (e.g. [8]–[12]). For our approach, we use the Jaccard index

$$\sigma_{\text{Jaccard}}(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}, \tag{1}$$

which is a simple metric widely used for computer code classification [13]–[19].

C. Related work

We conclude this section by highlighting related work.

Identification through network analysis: IoT device classification based on network analysis has been an active research topic for years. Meidan et al. [20] first trained an ML model to classify IoT devices using their generated network traffic with extremely high precision ($> 99\%$ accuracy). Since then, there have been many works (e.g. [21]–[24]) improving on the feature extraction algorithms used for network flow classification.

A major limiting factor of these classification approaches is the quality and scope of the datasets used to train their respective ML models. Jamali et al. [25] investigate the limitations of existing datasets and introduce a tool facilitating the collection of new datasets.

Active analysis: While the previously listed approaches do not interact with the analyzed devices beyond observing their network traffic, more active approaches exist. Lei et al. [26] propose an approach to classify IoT devices with web interfaces using features exposed through said web interfaces.

Individual device identification: Individual device identification, i.e. distinguishing between multiple devices of the exact same model, is a related yet distinct problem approached by, e.g., Sánchez et al. [27]. Such approaches primarily rely on hardware imperfections influencing measurable characteristics, such as clock speed and temperature.

To the best of our knowledge, no existing work attempts MCU architecture identification through static analysis of firmware. In contrast to all works referenced above, our approach does not make any use of machine-learning and relies only on public hardware description repositories and a simple similarity metric. Further, our focus lies on identifying the precise MCU architecture rather than producing information on the device's function or model.

Framework detection: Van Nielen et al. [28] propose a method to identify the framework used to develop firmware by inspecting strings contained in the produced firmware. Rhee et al. [29] develop a method to determine libraries used in

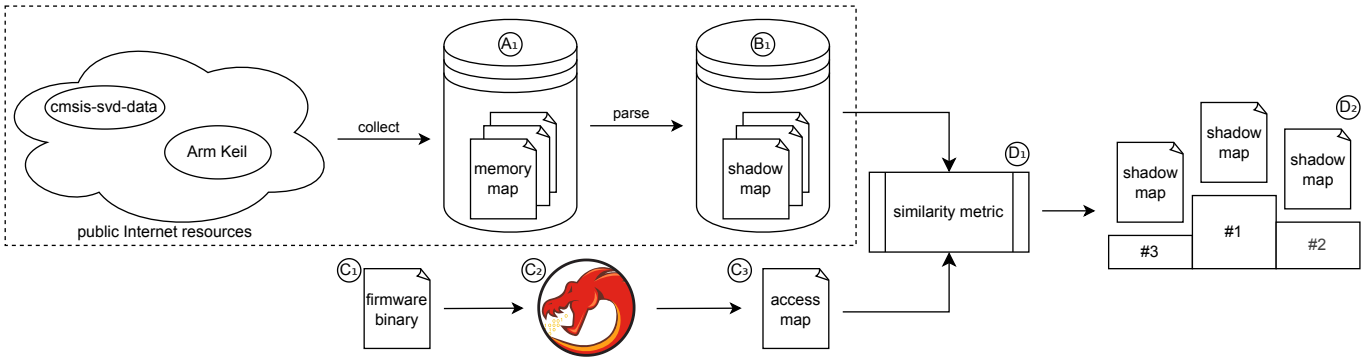

Fig. 2. An overview of our approach to identify the target MCU of a firmware binary. The dashed box contains pre-processing steps that are only run once: We collect a knowledge base (A1) of memory maps in the form of SVD files. These memory maps are parsed and translated into a knowledge base of shadow maps (B1) containing separate lists of all readable and writable addresses. We then identify the target MCU of a firmware binary (C1): We decompile and analyze the binary using Ghidra (C2) and use the analyzed binary to produce an access map (C3) containing separate lists of addresses read from and written to by the binary. Finally, we use a set similarity metric (D1) to produce a ranking (D2) of shadow maps based on their similarity to the access map.

application software by comparing the application binary to a knowledge base of known software. Adapting this approach to detect libraries used in firmware images could present an alternative to our approach.

Dataset generation: Hauser and Pennekamp [30] tackle the problem that large parts of device documentation are only available in PDF reference manuals. To this end, they develop a method to extract memory maps from PDF tables into SVD files.

III. Methodology

As shown in fig. 2, the basic idea behind our approach is to create an *access map* of addresses accessed by the firmware and to find the MCU in our knowledge base most similar to it. It can be broken down into four steps.

A. Collect machine-readable *memory maps* from public resources.
B. Construct a *shadow map*, i.e., a list of writable addresses and a list of readable addresses, from each *memory map*.
C. Construct an *access map*, i.e., a list of addresses written to and a list of addresses read from by the firmware binary.
D. Rank the constructed *shadow maps* by their similarity to the *access map*.

The first two steps result in a reusable knowledge base and only need to be done once[3]. The last steps need to be repeated for every binary. The remainder of this section will elaborate on these steps.

A. Collect memory maps

Our approach requires an extensive knowledge base[4] of device memory layouts. Building this knowledge base is essential for the final identification result, since we can only consider devices known to us when compiling a ranking of devices most fitting to the firmware. This first step builds such a knowledge base from public sources.

As detailed in section II-A, SVD repositories are a good resource to collect large numbers of machine-readable memory maps. We build our knowledge base of memory maps by retrieving all available SVD files from `cmsis-svd-data` and Arm Keil.

[3]Given the long and irregular update intervals of `cmsis-svd-data`, it should be sufficient to recalculate the knowledge base once a year.

[4]To avoid confusion, we want to emphasize at this point that the knowledge bases built in our approach are merely representations of pre-existing data and not trained ML models of any kind.

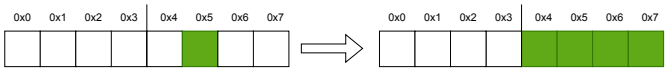

Fig. 3. A 1-byte register at `0x5` in the *memory map* results in the whole 4-byte word at `0x4` being stored in the *shadow map*.

B. Construct shadow maps

SVD files are created with the purpose of generating functional code, such as linker scripts and C header files. As such, they contain a large amount of information not relevant to us, such as names of registers and devices, the semantic meaning of individual bits, as well as the logical structure of registers and peripherals. In order to reduce the size of the knowledge base, we translate each memory map into a simplified *shadow map* that only keeps track of accessible addresses.

Sometimes, only parts of a memory-mapped register are intended to be readable. For example, a transmission register of a serial interface could have a read-only byte indicating the remaining space in the transmission buffer and three write-only bytes used to add data to the transmission buffer. One way to model such a register in an SVD file could be to define multiple one-byte registers. For such registers, it is not unusual for firmware to read all four bytes with a single 32-bit wide load instruction.

In order to avoid such register accesses not being correctly detected later, the shadow map intentionally over-approximates by only storing which 4-byte words contain at least one readable byte, instead of which precise bytes are readable (cf fig. 3). Similarly, if at least one byte in a word is writable, the whole word is stored in the list of writable addresses.

C. Construct access map

The goal of this step is to compile a list of addresses accessed by the firmware. We identify accessed addresses using static analysis. In particular, we use Ghidra[5] to perform the following steps on the decompiled firmware binary:

1) Consider the whole address space to be executable and readable but not writable.
2) Run all default analyzers on the program.
3) For every instruction containing a read or write reference to any address, round the referenced address down to the next 4-byte word (see fig. 3) and store it in the respective list in the shadow map.

D. Rank shadow maps

We compare the access map of our firmware image to every shadow map in our knowledge base. Our aim is to find the shadow map most similar to our access map, since that shadow map is most likely to represent the firmware's target MCU.

Let R_1 and W_1 be the sets of readable and writable addresses stored in a shadow map and let R_2 and W_2 be the sets of read and written addresses stored in the access map.

In order to find the shadow map representing the most likely target MCU, we rank the shadow maps using their composite score $\sigma_{\text{Jaccard}}(R_1, R_2) + \sigma_{\text{Jaccard}}(W_1, W_2)$. We consider the shadow map with the highest score to be the closest match.

IV. Evaluation

We evaluate the efficacy of our approach by considering the following research questions.

RQ1: How many MCUs can be distinguished using only information contained in the collected shadow map knowledge bases?
RQ2: How likely is the target MCU of a firmware image to be represented in our knowledge base?
RQ3: How often can our approach correctly identify the target MCU of a firmware image?
RQ4: How often can our approach correctly identify the target MCU series of a firmware image?
RQ5: How does changing the composition of the shadow map knowledge base affect the accuracy of our approach?

A. Ground truth and experimental setup

In order to evaluate the efficacy of our approach, we need a ground truth dataset of monolithic firmware images whose target MCUs are known and represented in our knowledge base. To answer RQ4, we are also interested in binaries for which only the target MCU series is represented in the knowledge base. We build our dataset from two sources:

Scharnowski et al. [5] have created a dataset to evaluate Fuzzware by compiling the same user application for 10 different target MCUs. These 10 distinct firmware images form the first part of our dataset.

Edge Impulse is a company developing a machine-learning platform that makes extensive use of IoT devices. To this end, they host 37 repositories on GitHub[6] containing monolithic firmware for 43 targets supported by their platform, of which 33 belong to MCU series represented in our knowledge base. Of these 33 targets, we were able to obtain 32 binaries, which form the second part of our dataset.

In order to systematically answer the research questions, we build three shadow map knowledge bases: One from Arm Keil, one from `cmsis-svd-data`, and one by combining both. For brevity, we will refer to these knowledge bases as B_{K}, B_{C}, and B_{L} respectively.

For every firmware image in our dataset, we note its correct target MCU, check which knowledge bases contain a representation of the MCU, and rank the shadow maps in all three knowledge bases using our approach.

B. Results

The complete evaluation results can be found in the artifact repository referenced in the appendix. We can summarize the results as follows.

RQ1: How many MCUs can be distinguished using only information contained in the collected shadow map knowledge bases?

At the time of evaluation, the knowledge bases B_{K} and B_{C} contain 3082 and 1738 total shadow maps, respectively. These numbers are bound to change whenever new SVD files are introduced to the upstream repositories.

In order for two MCUs in a knowledge base to be truly indistinguishable using only memory accesses performed by their firmware, their shadow maps must be identical. This is due to the fact that for any two shadow maps that differ in at least one address, there exists some possible access map for which these shadow maps would have different similarity scores. Hence, we answer RQ1 by evaluating how many MCUs in our knowledge bases have indistinguishable shadow maps.

The knowledge base B_{K} contains 1323 equivalence classes of indistinguishable shadow maps with the largest equivalence class comprising 163 identical shadow maps.

The knowledge base B_{C} contains 736 equivalence classes of indistinguishable shadow maps with the largest equivalence class comprising 131 identical shadow maps.

In both cases, the largest equivalence class contains only shadow maps describing a set of similar MCUs of the SiliconLabs series `EFR32xG` and `EFM32PG`.

RQ2: How likely is the target MCU of a firmware image to be represented in our knowledge base?

Together, the Fuzzware dataset and the Edge Impulse repositories contain firmware for 53 target MCUs. Of these targets, 36 (68%) are represented in B_{L} and 43 (81%) belong to an MCU series of which at least one device is represented in B_{L}.

RQ3: How often can our approach correctly identify the target MCU of a firmware image?

Using B_{L}, our approach identifies the correct MCU for 15/34 (44%) *identifiable*[7] binaries. The correct target MCU is contained in the top-3 results for 20/34 (59%) binaries.

[5]https://github.com/NationalSecurityAgency/ghidra

[6]https://github.com/edgeimpulse/

[7]i.e., binaries whose correct target MCU is represented in B_{L}

RQ4: How often can our approach correctly identify the target MCU series of a firmware image?

Using B_{L}, our approach identifies the correct MCU series for 24/42 (57%) binaries. The correct MCU series is contained in the top-3 results for 28/34 (67%) binaries.

RQ5: How does changing the composition of the shadow map knowledge base affect the accuracy of our approach?

The composition of the knowledge base has a strong effect on the accuracy of our approach. We can correctly identify more individual firmware images using the combined knowledge base B_{L} (15/34) than we can identify using only B_{C} (10/26) or B_{K} (10/24). However, since there are images that are correctly identified using B_{C} but incorrectly when using the larger B_{L}, choosing a larger knowledge base does not always yield better results. We investigate contributing factors by performing a case study on one such image here.

The access map of the firmware image `firmware-nordic-nrf5340dk.bin` lists 5173 accesses of which 368 (7%) are locations within the target MCU's RAM starting at address `0x2000 0000`. Using B_{C}, the correct shadow map `nrf5340_application` is ranked highest. Notably, none of the 15747 addresses listed in this shadow map begin with `0x2000 xxxx`, indicating that, as is common practice, the device's RAM was not included in the original SVD file.

Using B_{K}, the incorrect shadow map `BAT32G137A` is ranked highest. This shadow map lists 776 addresses, of which 352 (45%) lie between `0x2000 0000` and `0x2000 02c0`. While the listed addresses describe real memory-mapped peripheral devices, our approach mistakes the RAM accesses of our firmware image for accesses to these peripheral devices. The smaller size of the incorrect shadow map further increases its similarity score, resulting in it being ranked higher than the correct shadow map when considering the combined knowledge base B_{L}.

V. Discussion

In this section, we discuss the limitations of our approach and potential for future work.

The accuracy of our approach heavily depends on the quality of the used knowledge base. In particular, only MCUs that are represented in the knowledge base can be correctly identified. On the other hand, adding entries to the knowledge base can worsen the accuracy. Our case study indicates that references to global variables can be mistaken for peripheral accesses, leading to misidentification. It might be possible to improve the accuracy of our approach by identifying and removing references to RAM when creating the access map.

When creating access maps, our approach only considers memory accesses to fixed addresses. A more complete access map could be obtained using Fuzzware's [5] dynamic access modeling. Future work, could also verify the model derived by Fuzzware using the matched SVD description.

Finally, the CMSIS-SVD standard primarily targets MCUs based on Arm Cortex-A and Cortex-M architecture families. Accordingly, the SVD repositories used for building our knowledge base almost exclusively contain descriptions of Cortex-based microcontroller units. To mitigate this limitation, additional sources of memory map descriptions are needed.

VI. Conclusion

In this work, we presented a novel method to automatically identify the target MCU of monolithic IoT firmware by identifying which MCU described in the knowledge base matches the firmware's memory accesses most closely. To this end, we pre-calculated three different knowledge bases of MCU architectures from public sources and evaluated their influence on the accuracy of our method. The evaluation has shown that, using a knowledge base of sufficient quality, our approach correctly identifies the MCU series for 57% of evaluated binaries and finds the most precise available memory map for 44% of the firmware images in our dataset.

Acknowledgment

This research work was supported by the National Research Center for Applied Cybersecurity ATHENE. ATHENE is funded jointly by the German Federal Ministry of Research, Technology and Space and the Hessian Ministry of Science and Research, Arts and Culture.

This research work was supported by the German Federal Ministry of Research, Technology, and Space as part of the CASTLE project.

References

[1] W. Li, J. Shi, F. Li, J. Lin, W. Wang, and L. Guan, "μAFL: Non-intrusive feedback-driven fuzzing for microcontroller firmware," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 1–12.

[2] M. Eisele, D. Ebert, C. Huth, and A. Zeller, "Fuzzing Embedded Systems using Debug Interfaces," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, Jul. 2023, pp. 1031–1042.

[3] Z. Feng and J. Ma, "TWFuzz: Fuzzing Embedded Systems with Three Wires," in *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, ser. LCTES 2024. New York, NY, USA: Association for Computing Machinery, Jun. 2024, pp. 107–118.

[4] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, "HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1201–1218.

[5] T. Scharnowski, N. Bars, M. Schloegel, E. Gustafson, M. Muench, G. Vigna, C. Kruegel, T. Holz, and A. Abbasi, "Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 1239–1256.

[6] T. Scharnowski, S. Wörner, F. Buchmann, N. Bars, M. Schloegel, and T. Holz, "HOEDUR: Embedded firmware fuzzing using multi-stream inputs," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, Aug. 2023, pp. 2885–2902.

[7] "System View Description," https://arm-software.github.io/CMSIS_-5/SVD/html/index.html.

[8] O. Maqbool and H. Babri, "Hierarchical Clustering for Software Architecture Recovery," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 759–780, Nov. 2007.

[9] S. Fletcher and M. Z. Islam, "Comparing sets of patterns with the Jaccard index," *Australasian Journal of Information Systems*, vol. 22, Mar. 2018.

[10] SHC. Choi, S.-H. Cha, and C. Tappert, "A Survey of Binary Similarity and Distance Measures," *J. Syst. Cybern. Inf.*, vol. 8, Nov. 2009.

[11] R. Naseem, O. Maqbool, and S. Muhammad, "Improved Similarity Measures for Software Clustering," in *2011 15th European Conference on Software Maintenance and Reengineering*, Mar. 2011, pp. 45–54.
[12] H. Xia, B. Wang, Y. Zhang, and Y. Chen, "Hierarchical Clustering Combination Method Using Intuitionistic Fuzzy Similarity in Software Module Clustering," in *2023 International Conference on Networking and Network Applications (NaNA)*, Aug. 2023, pp. 502–507.
[13] S. Dolev, M. Ghanayim, A. Binun, S. Frenkel, and Y. S. Sun, "Relationship of Jaccard and edit distance in malware clustering and online identification (Extended abstract)," in *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, Oct. 2017, pp. 1–5.
[14] S. Lee, W. Jung, S. Kim, and E. T. Kim, "Android Malware Similarity Clustering using Method based Opcode Sequence and Jaccard Index," in *2019 International Conference on Information and Communication Technology Convergence (ICTC)*, Oct. 2019, pp. 178–183.
[15] J. Soni, N. Prabakar, and H. Upadhyay, "Behavioral Analysis of System Call Sequences Using LSTM Seq-Seq, Cosine Similarity and Jaccard Similarity for Real-Time Anomaly Detection," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, Dec. 2019, pp. 214–219.
[16] H. Faridi, S. Srinivasagopalan, and R. Verma, "Performance Evaluation of Features and Clustering Algorithms for Malware," in *2018 IEEE International Conference on Data Mining Workshops (ICDMW)*, Nov. 2018, pp. 13–22.
[17] B. P. Gond, M. Shahnawaz, Rajneekant, and D. P. Mohapatra, "NLP-Driven Malware Classification: A Jaccard Similarity Approach," in *2024 IEEE International Conference on Information Technology, Electronics and Intelligent Communication Systems (ICITEICS)*, Jun. 2024, pp. 1–8.
[18] H. Javed, F. Ali, B. Shah, and D. Kwak, "Binary Code Analysis for Cybersecurity: A Systematic Review of Forensic Techniques in Vulnerability Detection and Anti-Evasion Strategies," *IEEE Access*, vol. 13, pp. 167 139–167 164, 2025.
[19] Z. Luo, P. Wang, B. Wang, Y. Tang, W. Xie, X. Zhou, D. Liu, and K. Lu, "VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search," in *Proceedings 2023 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2023.
[20] Y. Meidan, M. Bohadana, A. Shabtai, J. D. Guarnizo, M. Ochoa, N. O. Tippenhauer, and Y. Elovici, "ProfilIoT: A machine learning approach for IoT device identification based on network traffic analysis," in *Proceedings of the Symposium on Applied Computing*. Marrakech Morocco: ACM, Apr. 2017, pp. 506–509.
[21] Z. Zhao, Y. Wang, Y. Lai, S. Yu, and J. Shi, "MUFFIN: A Meta-Knowledge Decoupling-Based Approach to Few-Shot IoT Traffic Classification," *IEEE Transactions on Networking*, vol. 34, pp. 411–426, 2026.
[22] A. Sivanathan, D. Mishra, S. Ruj, N. Fernandes, Q. Z. Sheng, M. Tran, B. Luo, D. Coscia, G. Batista, and H. H. Gharakaheili, "Real-Time and Trustworthy Classification of IoT Traffic Using Lightweight Deep Learning," *IEEE Transactions on Network Science and Engineering*, vol. 13, pp. 3256–3273, 2026.
[23] R. Taheri, J. Thom, N. Thom, B. Charyyev, E. Hand, and S. Sengupta, "Identifying Homogeneous IoT Devices With Hybrid Locality-Sensitive Hashing," *IEEE Internet of Things Journal*, vol. 12, no. 24, pp. 53 025–53 038, Dec. 2025.
[24] N. Zhang, S. Chen, S. Liu, X. Deng, Y. He, W. Xiang, and M. Li, "SecFinder: An IoT Device Identification System Based on Flow-level Traffic in Smart Home," in *2025 IEEE 31th International Conference on Parallel and Distributed Systems (ICPADS)*, Dec. 2025, pp. 1–8.
[25] A. F. Jamali, D. Rostami, and C. Fung, "IoT Device Identification using Deep Learning," in *2025 16th International Conference on Network of the Future (NoF)*, Sep. 2025, pp. 46–54.
[26] Z. Lei, Y. Li, Z. Li, X. Huang, D. Yu, N. Xue, and Y. Chen, "A fine-grained framework for online IoT device firmware identification via version evolution analysis," *Internet of Things*, vol. 34, p. 101767, Nov. 2025.
[27] P. M. Sánchez Sánchez, A. Huertas Celdrán, G. Bovet, and G. Martínez Pérez, "Adversarial attacks and defenses on ML- and hardware-based IoT device fingerprinting and identification," *Future Generation Computer Systems*, vol. 152, pp. 30–42, Mar. 2024.
[28] J. van Nielen, A. Peter, and A. Continella, "frameD: Toward Automated Identification of Embedded Frameworks in Firmware Images," in *Computer Security. ESORICS 2024 International Workshops*, J. Garcia-Alfaro, K. Barker, G. Navarro-Arribas, C. Pérez-Solà, S. Delgado-Segura, S. Katsikas, F. Cuppens, C. Lambrinoudakis, N. Cuppens-Boulahia, M. Pawlicki, and M. Choraś, Eds. Cham: Springer Nature Switzerland, 2025, pp. 514–533.
[29] J. J. Rhee, F. Zuo, C. Tompkins, J. Oh, and Y. R. Choe, "GrizzlyBay: Retroactive SBOM with Automated Identification for Legacy Binaries," in *MILCOM 2024 - 2024 IEEE Military Communications Conference (MILCOM)*, Oct. 2024, pp. 437–444.
[30] N. Hauser and J. Pennekamp, "[Tool] Automatically Extracting Hardware Descriptions from PDF Technical Documentation," *Journal of Systems Research*, vol. 3, no. 1, 2023.

APPENDIX

We make all artifacts needed to reproduce our evaluation, including software, pre-calculated knowledge bases, and firmware images, as well as detailed notes on all evaluation steps available in the following GitHub repository:

https://github.com/Fraunhofer-SIT/SDIoTSec2026-ortellius

In particular, the repository contains the knowledge bases of shadow maps used for our evaluation. These knowledge bases are accompanied by the software necessary to

- retrieve new SVDs from upstream repositories,
- rebuild the knowledge bases using the new SVDs,
- apply our identification method to any firmware binary and a knowledge base.