Poster: All Right Then, (Don't) Keep Your Secrets: Exposing API Hashing in Malware

Nicola Bottura[1], Giorgia Di Pietro[1], Yuya Yamada[2], Daniele Cono D'Elia[1], and Leonardo Querzoni[1]

[1] Sapienza University of Rome, Italy
{bottura,g.dipietro,delia,querzoni}@diag.uniroma1.it
[2] Nara Institute of Science and Technology, Japan
yamada.yuya.yy@gmail.com

Abstract. Modern malware employs disparate anti-analysis techniques to complicate analysis attempts. Among them, API hashing conceals the identity of imported library functions—key indicators for understanding malware behavior—by replacing their standard names with hashed values. Currently, resolving these obfuscated calls relies heavily on manual expertise and community-maintained hash repositories, both of which are time-consuming and difficult to scale. In this work, we explore an automated approach to deobfuscate API hashing. By leveraging dynamic program analysis, we identify and map hash values back to their original function names while also extracting information about the hashing scheme. Our method can then use malware itself as a "hash oracle", enabling on-demand resolution of standard function names through the malware's hashing logic, enabling automatic updates of repositories.

1 Introduction

When analyzing untrusted software for Windows platforms, the contents of the Import Address Table (IAT) allow analysts to make educated guesses about its capabilities. In fact, the identity of the APIs the executable references can provide valuable insights into the functionality of suspicious code. Beyond aiding in capability assessment, IAT content analysis also plays a key role in clustering malware samples and correlating threat group activities. As an example, the list of imported APIs has been used to track custom backdoors employed by specific actors [5]. Therefore, IAT contents are precious for static analysis efforts.

To hinder such analysis, malware authors have developed various API obfuscation schemes [2]. With dynamic API resolution, instead of being imported, functions are resolved only at run-time, typically using standard facilities like `LoadLibrary` and `GetProcAddress` [3]. However, these techniques are well-known to experts, and many modern analysis tools can disambiguate them.

A more advanced technique that significantly complicates static analysis is *API Hashing*. Instead of using standard resolution methods, which need the sample to materialize API names as strings in memory, malware can store hashed representations of these names and use these values to look up API addresses covertly. During execution, a sample scans all the functions exported by the loaded libraries, computing the hash of each symbol using a custom algorithm. These hashes are then compared against one or more pre-computed values. When a match is found, the desired function address has been solved. This complicates not only automatic analysis, but also manual reverse engineering attempts.

To combat API hashing, analysts often rely on their expertise and accumulated knowledge to recognize API hashing techniques. However, comprehensive documentation remains limited, with only a handful of practitioner blog posts in recent years offering insights through case studies [7,4]. A community-sourced online repository [6] maintains a collection of hashing algorithms observed in malware, along with precomputed hash tables for Windows APIs and other common strings. While helpful, this resource lacks automation and struggles to keep pace with the evolving landscape of hashing techniques, as even minor changes to a hashing scheme can render these tables obsolete. This variability—where readable API names are replaced with seemingly arbitrary hashes—impairs traditional reverse engineering approaches and highlights the growing need for automated solutions to support deep and accurate malware analysis.

This work explores a solution that enhances the analysis of malicious code without relying on manual investigation or expert intervention. Since API hashing obstructs static analysis and complicates accurate inspection of affected samples, our approach leverages dynamic program analysis to extract valuable information in a lightweight and generalizable manner. This can enable automated, scalable deobfuscation if the dynamic analysis comes with tenable costs. Along these lines, to support the continuous updating of community-maintained hash tables—and to provide analysts with more comprehensive data—we propose leveraging the extracted information to transform the malware itself into a "hash oracle". By emulating the sample's hashing logic and injecting arbitrary strings at the point where function names are typically resolved, this method allows for computing hashes using the malware's own code. This eliminates the need for manual reverse engineering to recover the API hashing logic.

2 Background

API hashing typically involves traversing the internal structures of Dynamic-Link Libraries (DLLs) to locate their export tables [3]. Once these tables are identified, the malware iterates through the list of exported function names, applying a custom hashing algorithm to each one. When a hash matches a precomputed target value, it indicates that the current API name corresponds to the obfuscated target function. At that point, the malware dynamically resolves the function's address by referencing its position in the export table, accessing the DLL's structure that maps function names to their corresponding addresses. A simplified version of this logic is provided in a snippet of C code in Listing 1.1.

Line 2 retrieves the name of the current API by using the DLL's base address and the name offset, while line 3 computes the hash of the API name using a custom algorithm. Line 5 checks whether the computed hash matches

```
1 for (i = 0; i < NumberOfFunctions; i++) {
2     api_name = (char *)(base + AddressOfNames[i]);
3     curr_hash = HASHING_FUNCTION(api_name);
4
5     if (curr_hash == pre_cmptd_hash) {
6         api_addr = (PDWORD)(base + AddressOfFunctions[
              AddressOfNameOrdinals[i]]);
7         return api_addr;
8     }
9 }
```

Listing 1.1. C code snippet for exemplary API hashing loop.

a precomputed hash; if a match is found, line 6 resolves the actual address of the obfuscated API. Notably, this method is not limited to API names: malware can apply a similar approach to obfuscate DLL names, traversing the Process Environment Block (PEB) to enumerate and hash loaded module names.

A core aspect of the API hashing technique is the use of *indicative offsets* from a DLL's base address to reference the offsets—Relative Virtual Addresses (RVAs) in Windows terminology—of key fields used for resolving API calls at runtime through API hashing. When deploying API hashing, malware writers typically maintain their own IAT-like array with the API addresses that they solve dynamically upon execution startup using the method above.

3 Proposed Approach

Manually reverse-engineering a binary that employs API hashing is a highly complex task. The presence of additional obfuscation layers can further complicate the process, making a complete analysis even more challenging. Even when no extra obfuscation is present and hash tables assist in deobfuscation, the process may still fall short: malware can implement customized or entirely new hashing algorithms that evade traditional analysis methods. To address this, we are currently exploring an automated solution that operates by inspecting accesses to DLL memory that are a precondition for any hashing mechanism.

As exemplified in Listing 1.1, API hashing relies on two key operations: (1) computing the API name, where the sample retrieves each plaintext function name from DLL memory, and (2) resolving the address of the obfuscated API name once found. Our approach focuses on identifying the instructions responsible for extracting these two critical information items. The code encompassing these two operations defines a *program slice* with the following properties:

1. The hashing computation occurs within this slice;
2. At some point, a successful match between the current hash and a precomputed one occurs (e.g., a `cmp` instruction with identical incoming operands);
3. The slice reaches its endpoint (symbol resolution) only after a found match.

Building on these properties, we developed a prototype solution that operates in three distinct phases (Figure 1), each needing some code (re-)execution.

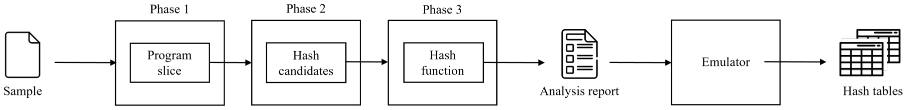

Fig. 1. Workflow of the proposed approach.

Phase 1: Extracting the API Hashing Program Slice. The first step focuses on identifying the code region responsible for the API hashing mechanism. This is achieved by detecting instructions that access DLL memory offsets viable to retrieve API names and addresses. By applying program slicing, we isolate a smaller, more relevant portion of the code, significantly narrowing the search space for hashing functions. This not only improves efficiency but also enables further analysis by leveraging the specific characteristics of the extracted slice.

Phase 2: Identifying Hash Candidates. With the program slice defined, the next step is to trace the operands of all comparison instructions within its boundaries. When a correct hash is eventually produced, execution reaches the slice's endpoint indicating a successful match. Although the exact hash value may still be unknown at this stage, we can postulate that one of the comparison operations in the slice must contain the correct hash as one of its operands, whereas the other operand will vary every time the samples moves to the next API name.

Phase 3: Identifying the Hash Function. Malware API hashing algorithms generally operate through a sequence of instructions that progressively modify a plaintext string, with the final iteration producing the hash value. To pinpoint the possible locations of the hashing function, we focus on arithmetic instructions commonly involved in these operations. To pursue generality, we do not assume that the hash value is returned by a function. On the contrary, by narrowing the scope to a smaller set of relevant instructions, we can map the resulting hash value back to the code sequence that generated it.

The insights gained from this analysis let us directly address a second major challenge: automating the community task of hash table creation and updating.

Currently, this process relies on manual reverse engineering to extract the hashing function, which is then submitted to the HashDB [6] service for updates. Due to the recurrent complexity of deobfuscating malware, analysts often resort to best-effort decompilation of the executable and sketch from there a Python implementation the original semantics for independent use. However, this method may incur the pitfalls of incorrect decompilation or manual follow-up translation. Additionally, it requires manual identification of hashing code.

We envision a solution that leverages the information gathered during the three phases above to orchestrate a controlled execution of the code in an emulator. This would let us use the malware itself as an on-demand hash value generator, collecting the hash value at each traversed DLL function symbol. This approach eliminates the need for costly reverse-engineering techniques to manage the hashing loop and to compute hash values.

		Phase 1		**Phase 2**	**Phase 3**	
Family	**Sample**	*Slice start*	*Slice end*	*Compare value*	*Hash value*	*Hash function*
BlackMatter	50c4970003a84cab1bf2634631fe39d7	✓	✓	✓	✓	✓
BlackMatter v2	d0512f2063cbd79fb0f770817cc81ab3	✓	✓	✓	✓	✓
Conti	bc92ea510a5630c770d9443be4b40fde	✓	✓	✓	✓	✓
Emotet	68c76c3403570a22ce7a60a1b68d9056	✓	✓	✓	✓	✗
Lockbit	628e4a77536859ffc2853005924db2ef	✗	✗	✗	✗	✗
Netwalker	73de5babf166f28dc81d6c2faa369379	✓	✓	✓	✓	✓
Revil	890a58f200dfff23165df9e1b088e58f	✓	✓	✓	✓	✗
Zloader	5c76c41f9d0cc939240b3101541b5475	✓	✓	✓	✓	✓

Table 1. Outcomes from sample analysis for the different stages of API hashing.

Discussion. In dynamic analysis scenarios, such as with a sandbox or antivirus, API interposition can reliably expose API call targets [3]. On the contrary, methods that rely on static information need other ways to address the challenges from API hashing. We try to bridge this gap using lightweight dynamic analysis.

The problem we investigate overlaps with what some program analysis techniques can offer. Taint analysis [8] can trace data from API names (sources) to hash comparisons (sinks), but its high overhead and susceptibility to imprecision—especially with malware code—limit its practicality. Input-to-state correspondence analysis [1] may be more tenable, but struggles when inputs do not directly map to resulting states. Hence, neither can directly untagle API hashing.

One limitation of our method lies in possibly large program slices when the hashing and comparison operations occur in separate loops. While this may impact performance, it does not reduce overall effectiveness, as the key instructions related to the hashing mechanism are still captured. Another challenging aspect may involve Phase 3, whenever multiple stack locations may be flagged as potential hash function sites (i.e., false positives). Despite this, when the correct hash value is identified in Phase 2, the method still supports effective analysis, even if the exact location of the hashing function remains uncertain.

4 Preliminary Results

To evaluate the feasibility of our approach, we conducted a preliminary assessment by developing a prototype implementation of our design, using the Dynamic Binary Instrumentation (DBI) capabilities of the DynamoRIO framework. The prototype covers the three phases of our method, whereas we are currently working on the emulation part to orchestrate executions from a snapshot.

Our initial testbed comprises eight samples drawn from popular families, such as *Emotet* and *NetWalker*, that notorioulsy utilize API hashing techniques. For each sample, we had access to auxiliary documentation and annotations from blog posts analyzing those specific samples, with a particular focus on their API hashing methods, enabling us to validate the accuracy of our prototype's output.

We consider a result successful if the prototype can: (1) correctly identify the relevant code slice, (2) locate the comparison instruction involving the hash values, and (3) recognize the underlying hash function.

The results are summarized in Table 1. Out of the eight samples, the prototype failed to produce any result only for *Lockbit*, due to a crash we suspect is related to a DynamoRIO bug and are further investigating. Among the remaining seven samples, the results were generally promising, with a few exceptions. In two cases, *Revil* and *Emotet*, the prototype failed to correctly identify the hash function. In both instances, the hash value was located within one of the instructions typically used by malware to transform the plaintext string during hashing that we look for in *Phase 3*. This is indicative that the hashing process is distributed across multiple functions not linked by internal calls from one another, presenting a challenge for our current analysis prototype. Since the hash value is not observed within a single isolated function, our prototype cannot reliably identify the complete hashing logic. This limitation warrants further investigation. On the bright side, for both versions of *BlackMatter*, which performs API hashing also on DLL names, our prototype operated successfully in full.

5 Conclusion

We have proposed an automated approach for analyzing API hashing mechanisms in Windows malware. Once mature, our prototype may remarkably ease ongoing community-driven efforts in this domain. As a next step, we aim to extend it by addressing its current limitations in the identification of the hash function and evaluating its performance on a larger set of malware samples.

This work has been partially supported by projects SERICS (PE00000014) and Rome Technopole (ECS00000024) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

1. Aschermann, C., Schumilo, S., Blazytko, T., Gawlik, R., Holz, T.: REDQUEEN: Fuzzing with input-to-state correspondence. In: NDSS (2019)
2. Cheng, B., Ming, J., Leal, E.A., Zhang, H., Fu, J., Peng, G., Marion, J.Y.: Obfuscation-Resilient executable payload extraction from packed malware. In: USENIX Security Symposium. pp. 3451–3468 (2021)
3. D'Elia, D.C., Nicchi, S., Mariani, M., Marini, M., Palmaro, F.: Designing robust API monitoring solutions. IEEE Transactions on Dependable and Secure Computing **20**(1), 392–406 (2023)
4. Gupta, N.: API hashing - why malware loves (and you should care), `https://securitymaven.medium.com/api-hashing-why-malware-loves-and-you-should-care-77c5135d9aaa`
5. Mandiant: Tracking malware with import hashing, `https://cloud.google.com/blog/topics/threat-intelligence/tracking-malware-import-hashing/`
6. OALabs: Hash db, `https://github.com/OALabs/hashdb`
7. Red Team Notes: Windows API hashing in malware, `https://www.ired.team/offensive-security/defense-evasion/windows-api-hashing-in-malware`
8. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In: IEEE Symposium on Security and Privacy. pp. 317–331 (2010)