Excel@FIT 2015
http://excel.fit.vutbr.cz

Fast Cryptographic Constants Identification in Retargetable Machine-code Decompilation

Peter Matula*

Abstract
Retargetable machine-code decompilation transforms a platform-independent executable into a high level language (HLL). Decompilers may be used by reverse engineers to manually inspect suspicious binaries (e.g. malicious software). However, modern malware is often very complex and its decompilation produces huge and enigmatic outputs. Therefore, reverse engineers are trying to use every bit of available information to aid their analysis. This paper presents a fast cryptographic constants searching algorithm. Based on a signature database, it recognises usage of constants typical for many well-known (cryptographic, compression, etc.) algorithms. Moreover, the results are used by AVG Retargetable Decompiler to make its output more readable. The presented algorithm is faster than any state-of-the-art solution and it is fully exploited in the decompilation process.

Keywords: reverse engineering — decompilation — data type recovery — cryptographic constants

Supplementary Material: Downloadable Test Suite

*imatula@fit.vutbr.cz, *DIFS, Faculty of Information Technology, Brno University of Technology*

1. Introduction

Retargetable decompiler is a reverse engineering tool which transforms platform-independent input binary into an output high level language (HLL). It is often used alongside disassembler to manually analyse potentially malicious binaries. Even though decompiler's output is generally more readable than disassembled symbolic instructions, it can still be huge and enigmatic. In such a case it is desirable to employ any analysis that helps the user to minimise his effort. One such analysis is a cryptographic constants identification offering the following profits:

- It identifies data with known semantics.
- It assigns known types to such data.
- It can track usages of such data.

The task of a constants identification analysis is to search the binary for data sequences used by well-known algorithms. These are typically cryptographic, compression or other similar methods[1] which always use prescribed constants (e.g. S-boxes, polynomials).

[1]In this article, they are all referred to as cryptographic.

Such algorithms are often used by malware and their identification can significantly ease up the analysis. The sequence of constants in executable is recognised by 1:1 comparison with all signatures from the specialised database (DB, Section 2). Since both DB and binary may be huge, search must be fast but reliable. Signatures can be divided into:

- Fixed-length signatures (Section 3).
- Variable-length signatures (Section 4).

The idea of searching the binary for known sequences is not new. In fact there are several existing tools that analyse binary files in this manner. Older tools like *CryptoChecker*, *CryptoSearcher* or *Kanal* [1] have very limited DBs embedded into their source codes. The newer ones like *Signsrch* [2], *ClamAV* [3], *Balduzard* [4] or *YARA* [5] rely on massive external DBs containing thousands of signatures. Upon successful detection, all of them print identified signature's information and its offset in binary file. This is better than nothing, but still not very convenient for the analyst. Only the *FindCrypt* [6] plugin to *IDA disassembler* [7] interacts with disassembled instructions.

It's embedded DB is however fairly limited (dozens of signatures), it can not be easily extended and it is unknown how well it would scale if huge Signsrch-like database was used.

In our solution, an already existing DB is preprocessed and embedded into our sources to minimise its run-time parsing (Section 2), while keeping the option of user added signatures. Then, an algorithm based on the pre-existing general idea, but designed for fast signature search and minimal number of comparisons, scans the input binary (Sections 3, 4). Finally, results are integrated into the decompiler's analyses 5.

In this paper, we explore several possible search solutions and combine their best aspects into the fastest cryptographic constants search algorithm (Section 6). Moreover, we propose a novel application of collected results in the AVG Retargetable Decompiler framework.

Note: Section 6 shows only the experiments comparing our final solution with other state-of-the-art detectors. The experiments supporting the design of our search algorithm are placed closer to the discussed ideas in Sections 3 and 4. They were carried out under the same conditions as experiments in Section 6.

2. Cryptographic Signature Database

Database of over 2000 signatures used by Signsrch have become the standard for all the tools of similar orientation. Single signature consists of its name, data type and an array of constants for which the input binary is searched. DB used by ClamAV has a different format, but roughly the same contents, since it was created from the original Signsrch DB. Furthermore, ClamAV's DB can be converted into YARA rules which can be fed to Balduzard or even YARA itself. We chose to use YARA database in our project, because the format is well established, widely used and more general than the other two. Therefore, we can use it in the future to create all kinds of more complex (not necessarily cryptographic) signatures.

We also decided to combine the advantages of a huge external DB with the efficiency of embedded DB. Original YARA rules were preprocessed and converted into a C++ code which initialise internal DB containers in the most efficient way. Therefore, the program no longer needs to parse and preprocess external DB every time it runs. However, the program can still be provided with an additional, user created, DB. Time saved in a single run is not that important, but as is shown in Figure 1, it gets very significant when many decompilations are run.

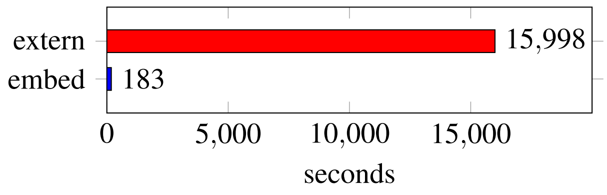

Figure 1. Overall time needed to parse/load an external/embedded DB in over 60,000 decompilations from our vast internal test suite.

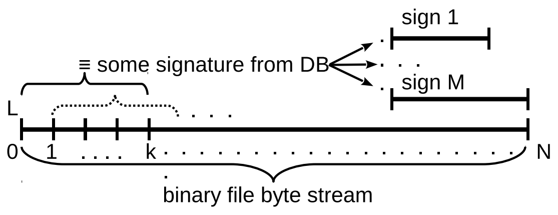

Figure 2. Naive search principle. All signatures are compared with all locations.

3. Fast Fixed-length Signature Search

This section presents a gradual build up from the naive fixed-length signature searching to the algorithm several times faster than any state-of-the-art solution (see Section 6). A fixed-length signature contains a fixed-length constant array, which can be represented as a hexadecimal string (e.g. `9410201f5ba70bef`) denoting a complete sequence of array's bytes. Our DB currently contains 2192 such signatures.

3.1 Naive Approach

Naive approach to the fixed-length signature search is depicted in Figure 2. Input binary file is traversed byte by byte and all signatures are tried to by applied on every location L. *Application* means, that signature's constant array of length k is compared with $L_0, \ldots, L_{k-1}$ bytes from the input byte stream. If they fully match, a cryptographic constant sequence is successfully identified. The matching is implemented by byte comparison. We also experimented with the hash comparison, but it proved to be error-prone on huge binaries.

3.2 Search Range Reduction

Tools presented in Section 1, as well as the previous naive approach, scan the whole binary. However, executables are divided into segments, only some of which are loaded and executed. Unloaded segments can not affect program's execution and are therefore useless from this analysis point of view. The number of comparisons can be further limited by setting the comparison step to the minimal alignment used by architectures supported by the decompiler. The effects of such limitations are shown in Figure 5.

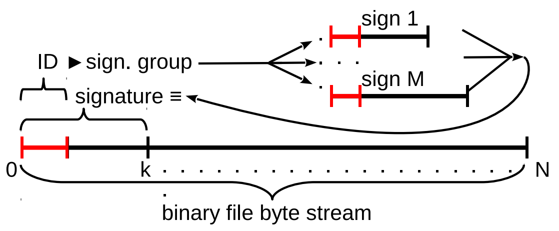

Figure 3. Prefix-based search principle. Signatures are selected by an ID computed from their prefixes.

3.3 Prefix-based Search

Trying all signatures on all locations is far from effective. Therefore, we divide signatures into groups based on IDs created from their prefixes. Signature is tried to be applied only if the current location matches its prefix (Figure 3). High efficiency is achieved by choosing the best prefix length — 1) it must be short enough to be easily comparable, 2) but long enough to maximize ID uniqueness. Even though multiple signature comparisons can still be triggered on a single location, Figure 5 proves that an impressive speedup was achieved.

3.4 ID Collision and Prefix Hit Reduction

Prefix hit occurs each time a possible signature prefix is found, ID computed, and the comparison triggered. If two or more signatures have the same ID, we say they are in an *ID collision*. In such a case, a single prefix hit causes several signature comparisons. To speed up the prefix-based search, we need to reduce occurrences of both of these phenomena.

One way to reduce ID collisions is to increase the prefix size. Longer prefix implies lower collision probability. However, it also makes all prefix operations more complicated (i.e. slower). Therefore, we prefer the approach shown in Figure 4. It computes an ID from any suitable chunk of the signature. Since signatures are fixed-length, it is possible to exactly calculate chunk's offset from the beginning. The offset can then be used to start the comparison at the correct location, even when ID was computed later in the byte stream. This gives us many possible IDs for every signature and therefore more room to pick a unique one. Figure 6 proves the positive effect of this step on the structure of our database.

To reduce the number of prefix hits, we further exploit the possibility to pick a signature's ID from many different options. By simply choosing the one with the highest entropy, we greatly reduce the number of performed comparisons (see Figure 7).

By putting it all together, we get nearly twice as fast fixed-length signature search algorithm than the prefix-based method (see Figure 8).

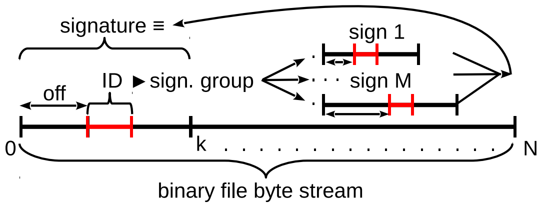

Figure 4. ID collision reduction search principle. Signatures are selected by an ID computed from any suitable chunk. Each signature gets its own offset. Offsets may differ even for signatures within the same ID group.

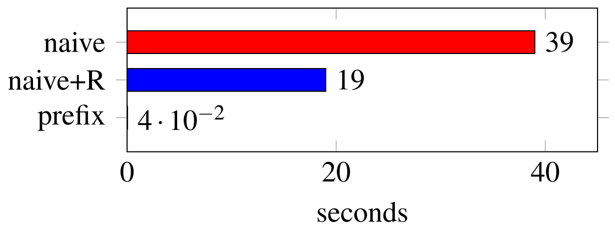

Figure 5. Running time comparison of naive approach (naive), naive approach with search range reduction (naive+R), and prefix-based search (prefix) on `naive-test` example from the downloadable test suite.

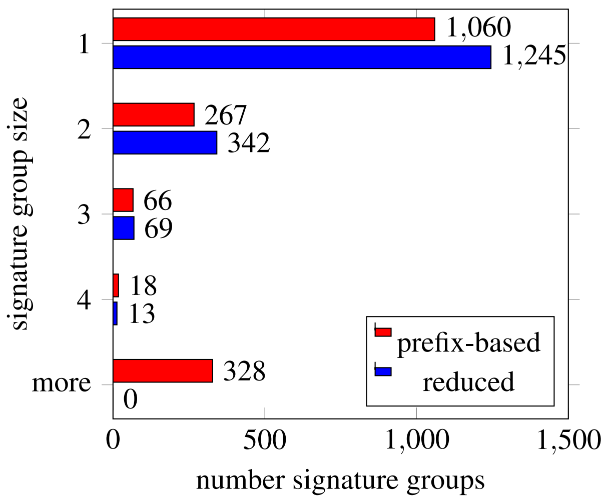

Figure 6. Database structure — how many and how big are the signature groups. The smaller groups the better. In the prefix-based search, some groups contain up to 98 signatures.

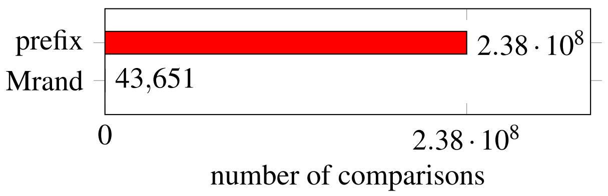

Figure 7. The number of performed signature comparisons using the prefix-based IDs (prefix) and the highest entropy IDs (Mrand) on `all-tests` archive from the downloadable test suite.

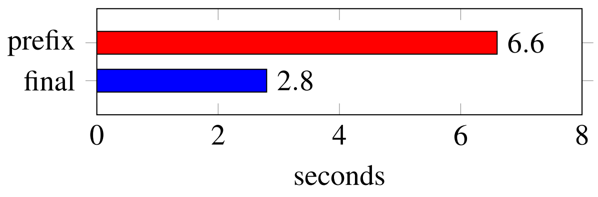

Figure 8. Running time comparison of prefix-based search (prefix) and the final fixed-length algorithm (final) on `all-tests` archive from the downloadable test suite.

4. Fast Variable-length Signature Search

This section presents a principle behind the variable-length signature searching and proposes an improved method inspired by the techniques shown in the previous section. A variable-length signature contains a constant array which consists of alternating fixed and variable parts. It can be represented as a hexadecimal string with the construction `[X-Y]`. For example, `c6ef3720[0-20]61c88647` denotes an array starting with the fixed sequence `c6ef3720`, followed by any arbitrary sequence from 0 to 20 bytes long, followed by `61c88647`. Our DB currently contains 95 such signatures.

4.1 Naive Approach

Naive approach to the variable-length signature search is similar to the fixed-length one depicted in Figure 2. All variable-length signatures start with a fixed-length prefix which can be used to trigger the signature matching. Fixed parts are matched by byte comparison. Variable parts' bytes are skipped until the next fixed chunk is reached or the maximal length is exceeded. If the last fixed part matches, the whole signature can be applied.

4.2 Fast Variable-length Search

Fast variable-length signature search is inspired by Section 3.4. Since variable-length signature matching is clearly more time consuming than the fixed-length one, we prioritize variable signatures and assign them the most unique and the highest entropy IDs possible. This greatly reduces the number of failed application attempts. Comparison algorithm itself is a naive search modification which takes ID's position in signature into account. Thanks to this modification and the small number of variable-length signatures, performing a variable-length search alongside the fixed-length detection does not dramatically increase algorithm's time complexity (see Figure 9).

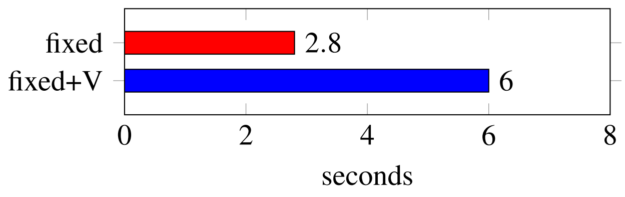

Figure 9. Running time comparison of final fixed-length search without (fixed) and with (fixed+V) variable-length search on `variable-test` example from the downloadable test suite.

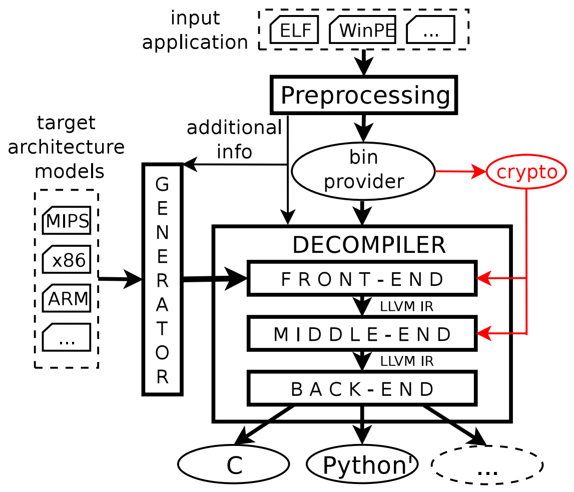

Figure 10. AVG Retargetable Decompiler concept.

5. Integration into AVG Retargetable Decompiler

Along with the significant signature identification speed-up, the main ability that sets our solution apart is its integration into the AVG Retargetable Decompiler framework [8]. The output of our cryptographic analysis is not just a list of offsets into binary file. The gathered information is fully utilized by the decompiler to produce more readable output code.

5.1 AVG Retargetable Decompiler Structure

AVG Retargetable Decompiler aims to be independent on any particular architecture, operating system, compiler or object file format. Thanks to the ADL processor models and an extensive preprocessing, it is able to decode machine-code into an architecture independent LLVM IR. Its abstraction level is then lifted by a sequence of analyses and the final HLL is produced. Decompiler's basic concept is depicted in Figure 10, more detailed description can be found in [9].

Figure 10 also shows in red the placement of the new cryptographic constants analysis. It runs right after preprocessing and passes its results to the decompiler's front and middle ends.

5.2 Improving Decompiler's Results

The output of cryptographic constants analysis is a list of locations (addresses) and signature entries associated with them. Front and middle end analyses can access this information and use it to their own benefit. Global variable analysis sets more readable names and precise data types to the identified objects. Type recovery analysis propagates these types with priority (see [10, 11]). Function analysis checks its instructions and rename itself (only if it does not already have a meaningful name) if they are operating on a constant array. Finally, the code generator makes sure objects are initialised with the proper numerical constants. The difference these steps make can be observed in Figure 11 and Figure 12.

```
int global_0x8049780 = 0x00000000;

int function_0x80483cb()
{
  for (int i=0; i<256; ++i)
    printf("%d\n",
      *(4 * i + &global_0x8049780) );
}
```

Figure 11. Decompiled code of a function printing the CRC table without the cryptographic analysis.

```
unsigned int crc_32_tab[] = { 0x00000000,
  0x77073096, /* ... */, 0x2d02ef8d }

int crc_32_tab_user_1()
{
  for (int i=0; i<256; ++i)
    printf("%d\n", crc_32_tab[i]);
}
```

Figure 12. Decompiled code of a function printing the CRC table with the cryptographic analysis.

6. Experiments

This section presents the experiments we performed on our cryptographic constants detection analysis. It focuses on a running time speedup of our solution compared to other state-of-the-art detectors. We do not test detection success rate, since it is perfect if expected constant arrays are present in the binary, and zero if they are not. If for example a program computes AES encryption without using expected data arrays (e.g. they can be obfuscated or dynamically calculated), this kind of analysis has no chance to succeed. The experiments were carried out on a personal computer equipped with Intel Xeon E5-1620 3.70GHz CPU and 16 GB of RAM. The test suite contains few dozens of binaries for several different architectures. Its total size is 130 MB. All binaries are real-world malware and most of them contain some detectable cryptographic constants. Test suite, README and a 64-bit Linux binary of our analysis can be obtained via the active link in the "Supplementary Material" below the abstract.

Experiment's results shown in Figure 13 prove that, while using roughly the same-sized DB, our algorithm is 5 to 18 times faster than other state-of-the-art detectors.

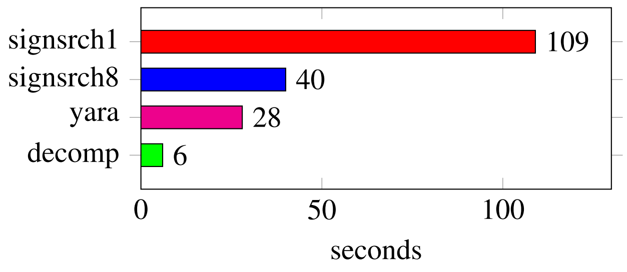

Figure 13. Running time comparison of our full (fixed and variable-length) cryptographic constants identification analysis (decomp) and several state-of-the-art detectors. Signsrch tool was run twice, using one (signsrch1) and eight (signsrch8) threads. All the other detectors use only one thread. Experiments were carried out on `all-tests` archive from the downloadable test suite. All of the detectors are using roughly the same sized (but different format) signature databases.

7. Conclusions

In this paper, we presented a fast cryptographic constants identification algorithm operating on binary files of any target architecture. We also showed how to integrate its results into AVG Retargetable Decompiler and the benefits it has on the HLL output quality.

Our solution turned out to be 5 to 18 times faster than other state-of-the-art detectors. It also offers an added value in form of a quality HLL code which is more helpful to the reverse engineer than a simple list of detected constants and their file offsets.

This work proposes several naive search algorithm improvements which allows much more efficient signature searching. It also introduces a novel idea of embedding such an algorithm into a decompiler and use its results to modify the generated code.

Our main goal for the future is to make the analysis more general and use it for other problems similar to the cryptographic constant searching. AVG Retargetable Decompiler already performs two signature based binary searches: 1) a generic statically linked code detection [12], and 2) a compiler detection [13]. Both of them use their own signature format and both are significantly slower than search method presented in this article. Despite a few peculiarities, it would certainly be possible to unify their signatures via YARA format and exploit the efficient search principles described in this paper. Ultimately, the YARA rules may be used to describe all kinds of more complex patterns.

As was already mentioned in Section 6, this kind of analysis has no chance to succeed if the cryptographic constants are obfuscated or dynamically computed (i.e. the expected byte arrays are not in the data section). In that case, much more complicated analyses inspecting functions' semantics have to be employed [14, 15]. One of our future goals may also be a retargetable implementation of such an analysis.

Acknowledgement

This work was supported by the BUT FIT grant FIT-S-14-2299, and by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

References

[1] KANAL - Krypto Analyzer for PEiD. ftp://ftp.lightspeedsystems.com/RyanS/utilities/PEiD/plugins/kanal.htm. [cit. April 27, 2015].

[2] Signsrch. http://aluigi.altervista.org/mytoolz.htm. [cit. April 27, 2015].

[3] ClamAV. http://www.clamav.net/index.html. [cit. April 27, 2015].

[4] Balbuzard. http://www.decalage.info/en/python/balbuzard. [cit. April 27, 2015].

[5] YARA. http://plusvic.github.io/yara/. [cit. April 27, 2015].

[6] FindCrypt. http://www.hexblog.com/?p=27. [cit. April 27, 2015].

[7] IDA Disassembler. www.hex-rays.com/products/ida/, 2012.

[8] Retargetable decompiler. https://retdec.com. [cit. April 27, 2015].

[9] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. *International Journal of Security and Its Applications (IJSIA)*, 5(4):91–106, 2011.

[10] Peter Matula and Dušan Kolář. Reconstruction of simple data types in decompilation. In *Sborník příspěvků Mezinárodní Masarykovy konference pro doktorandy a mladé vědecké pracovníky 2013*, pages 1–10. Akademické sdružení MAGNANIMITAS Assn., 2013.

[11] Peter Matula and Dušan Kolář. Composite data type recovery in a retargetable decompilation. In *Proceedings of the 9th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 63–76. NOVPRESS s.r.o., 2014.

[12] Lukáš Ďurfina and Dušan Kolář. Generic detection of the statically linked code. In *Proceedings of the Twelfth International Conference on Informatics INFORMATICS 2013*, pages 157–161. Faculty of Electrical Engineering and Informatics, University of Technology Košice, 2013.

[13] Jakub Křoustek, Peter Matula, Dušan Kolář, and Milan Zavoral. Advanced preprocessing of binary executable files and its usage in retargetable decompilation. *International Journal on Advances in Software*, 7(1):112–122, 2014.

[14] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot: Cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 169–182, New York, NY, USA, 2012. ACM.

[15] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated identification of cryptographic primitives in binary programs. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 41–60, Berlin, Heidelberg, 2011. Springer-Verlag.