A Vision for Enhancing Security of Cryptography in Executables

Otto Brechelmacher, Willibald Krenn, and Thorsten Tarrach(✉)

AIT Austrian Institute of Technology, Giefinggasse 4, 1210 Vienna, Austria
thorsten.tarrach@ait.ac.at

Abstract. This paper proposes an idea on how to use existing techniques from late stage software customization to improve the security of software employing cryptographic functions. In our vision, we can verify an implemented algorithm and replace it with a faster or more trusted implementation if necessary. We also want to be able to add encryption to binaries that currently do not employ any, or gain access to unencrypted data if an application depends on encryption.

To corroborate the feasibility of our vision, we developed a prototype that is able to identify cryptographic functions in highly optimized binary code and tests the identified functions for functional correctness, potentially also revealing backdoors.

1 Introduction

In recent years cryptography became almost universal in applications, both commercial and open-source. Every application securely communicating with its cloud server relies on a full suite of symmetric and asymmetric encryption algorithms. Hence, running a closed-source application employing cryptography requires one to trust the software developer to (a) consistently use cryptography, (b) have chosen a good crypto-library and use it correctly, and (c) not to have built-in a backdoor that leaks sensitive information.

Especially when dealing with software running in a sensitive environment it is necessary to thoroughly inspect the software for potential vulnerabilities and threats. We focus on the correctness analysis of the software using the binary code as unwanted functionality can be introduced during the compilation process [1], the original source code or libraries are not available, or one cannot re-produce bit-matching binary images from the supplied sources. For similar reasons, i.e. no source available or highest level of trust, any late stage customization has to be performed on the machine-code level. In the following, we focus on cryptography routines because they are often exposed to the network and present a major part of the attack surface of an application. Also, this specific domain allows us to use a lot of domain-specific knowledge in the attempted automated analysis.

The research leading to this paper has received funding from the AMASS project (H2020-ECSEL no. 692474).

M. Payer et al. (Eds.): ESSoS 2018, LNCS 10953, pp. 1–8, 2018.
https://doi.org/10.1007/978-3-319-94496-8_1

Once some piece of software fails the verification, information about vulnerabilities becomes public, or encryption needs to be added due to changing security requirements, the software needs to be patched or replaced. In order to save on the re-certification/verification effort, we propose to automatically address the shortcomings in the binary file. Put differently, we do not only want to analyze binaries, but also fix certain problems automatically or even add encryption to binaries that do not currently employ encryption, e.g. adding HTTPS to a legacy application.

While one source of vulnerabilities are sophisticated, deliberate backdoors, we believe a significant portion of flaws are accidental. Developers typically are not very knowledgeable in cryptography. A common pattern these days is that developers use the first seemingly working code-snipped from some blog or stack-exchange answer for their code. There are numerous problems when using cryptography in this naïve way.

The first problem is that developers may use a home-grown implementation of their desired cipher or some not very well maintained library. Such home-grown implementations will not contain hardening against side-channel attacks and will likely contain unknown bugs or weaknesses [2]. The second problem is connected to the use of statically linked libraries. Here, one issue is that the user of the software cannot easily update the version of the library used, once a vulnerability has been found. Further, the user has no guarantees (and is generally unable to check) whether the software supplier has used the latest version of the library when building the release. As security issues are regularly found and fixed in libraries, this is a major problem. Lastly, even the best crypto library may be used incorrectly. There are numerous topics around encryption, such as key management or safe storage and erasure of the key in memory that a developer can get wrong.

In the past numerous flaws concerning cryptography have been found in software. Here are some of the more interesting cases. Our first example has been revealed by the Chaos Computer Club of Germany (CCC) in software used by the German government to spy on suspected criminals. On close analysis of the software, which is also known as the federal trojan horse, the CCC discovered that the trojan used the same hard-coded AES-key to secure the data transfer to the control server for years. The key can now be found on the internet [3]. Hence, unauthorized parties would have been able read the data transferred. The second example is the well-known Heartbleed bug [4]. This bug may still be present in applications that were statically linked with any of the vulnerable OpenSSL versions. Here, we would want an automated upgrade of the library linked into the affected applications to fix the issue. A third example shows another source for weaknesses when using cryptographic functions: the random number generator. The best crypto implementation does not help if the key is predictable. This was the case in Debian's OpenSSL package, that was able to generate only 32 768 different keys [5]. As before, we would like to be able to address this bug by replacing the random generator used in the application by a trusted one. These are just a few examples, but they illustrate a need to focus on cryptography as part of a software security analysis and life cycle. They also

show that binary analysis is needed because merely analyzing the input and output of the application may not reveal a faulty crypto implementation.

We are now ready to present our vision in Sect. 2, before arguing for its feasibility in Sect. 3 by presenting already existing work we can build on. Finally, we conclude this idea paper in Sect. 4.

2 The Vision

As we have shown in Sect. 1, the current treatment of cryptography in the life-cycle of executable programs is far from optimal. Put shortly, the user has no efficient way of checking whether the employed algorithms or library versions are up-to-date, whether the cryptographic primitives are used correctly, or whether the advertised algorithms are used at all. Even worse, once vulnerabilities are discovered there is no easy way to fix the affected executables. Finally, there is no way to add additional functionality to the application.

Our vision is to address these shortcomings with a platform for analyzing, rewriting, and embedding cryptographic functions in binaries. By 'binary' we mean a compiled program, where we don't have access to the source code or any debug information. We understand the term cryptography here in its wider sense to also include random number generators needed to generate keys, cipher modes (counter mode, cipher feedback mode, etc.), and cryptographic protocols like TLS. We envision our platform to have the following features:

- **Analyze** cryptographic functions in the binary for correctness and potential backdoors. The analysis should reveal if the algorithm computes the correct result, has potential information leaks, and if it is vulnerable to known attacks.
- **Replace** cryptographic functions in the binary with alternate implementations.
- **Insert** cryptographic functionality, which is most useful for legacy binaries.

While these features seem trivial, achieving them is far from it. We will use a few scenarios to illustrate common problems and how our platform would help.

Unknown Binary Using Crypto. Assuming we bought an application that employs the Advanced Encryption Standard (AES) according to the data sheet. In this situation we need to check whether AES really is being used and whether there are (known) vulnerabilities, weaknesses, or leaks with this version. Hence in this scenario we could use the analysis capabilities of our platform. In case we are not satisfied with the quality of the implementation, we can then replace the existing AES implementation with calls to our own trusted library.

Legacy Application Without Patches. Another important situation is the maintenance of legacy applications without vendor support. Be it a statically linked application or an application dynamically linked to some outdated version

of some cryptographic library that contains known vulnerabilities. In order to develop our own patch for the application, our platform can be used. In the case of an statically linked application, the platform will supply the means to replace the original library version with an updated one. If the application has a dynamic dependency one might be able to find a drop-in replacement if the interface did not change. If, however, there was some interface change, our platform will help with its insert and replacement functionality to add adapter code so the application can use the interface incompatible new version of the library.

Adding Encryption. A further use case we envision for our platform is to add encryption to executables. This could be done by adding transient encryption/decryption when saving/loading files but could also mean securing network connections in non-standard ways. For example we may want to enable a legacy application to encrypt its network traffic. The encryption could be added to the data passed to the send function of libc and the decryption to the data returned by the receive function. We could provide standard packages that also take care of key derivation between peer, for example as part of the connect call. Similarly employing steganography or adding direct support for onion routing techniques would benefit from our platform.

Weakening Encryption. While not in our primary focus, our platform could even be used to weaken a cryptographic implementation. This could be useful in case of reverse engineering, i.e. malware analysis. An important part of reverse engineering is the network traffic, which may be difficult to analyze if it is TLS encrypted with certificate pinning. Our framework would save the analyst a lot of time by simply leaking the encryption key to a file.

All of the discussed scenarios can be realized manually with dedicated personnel. Our platform, however, should automate the work involved as much as possible. While this is no easy feat, we think it is viable and give an overview of already available building blocks in the following section.

3 Available Already

Quite a number of building blocks for our vision are already in place. On the one hand there is a large body of research and tools that deal with binary analysis and manipulation, on the other hand we started working towards our platform for Linux/x86-64 and gained first, encouraging results.

In the following sub-sections, we give a brief overview of available tools and techniques that help realizing our vision. Starting from tools helping with binary rewriting and analysis, we refer to techniques used for specifying machine code that needs replacement before describing our own contributions.

3.1 Supporting Tools

Binary Rewriting. We benefit from a large body of work in runtime manipulation of machine code and binary rewriting. We use DynamoRIO [6] to inject code and modify the control flow of binaries during runtime. While this is ideal for prototyping we would eventually want to rewrite the binaries to persist changes in it. Rewriting binaries is challenging because the control flow can be unpredictable due to exceptions and signals and because some parts of the code may be only revealed at runtime due to Just-In-Time compilation or encryption. Thankfully many of these problem are already addressed, e.g. in Zipr++ [7] and RL-Bin [8].

Binary Analysis. There are two approaches to analyzing a binary. One is to observe the binary during its normal operation at runtime. That is suitable to understand the normal operation of the binary, but not to find a backdoor. The latter can be found for example by symbolic analysis, a technique to reach all program locations. The downside is that such analysis is inherently slow. Since we expect the crypto routines to be part of the normal operation of the binary our prototype uses runtime analysis. We use DynamoRIO to record a trace that we later analyze. A trace means the sequence of all machine code instructions that were executed during a single run of the binary under observation.

There are a few symbolic analysis frameworks for machine code, including S2E [9], angr [10], and libtriton [11]. We also showed that it is feasible to use KLEE [12] for symbolic analysis if the machine code is first lifted to LLVM intermediate language. A task that is non-trivial in itself and we completed only for simple binaries.

Specification. We specify functions and their input/output behavior with the help of model programs. That means we have a model of the intended functionality as a specification and are searching for functions that, when given the same input, will return the same value. Hence our specification does not use pairs of inputs and outputs, but a reference implementation of the function we are looking for. To our knowledge this specification approach is novel in the field.

Another specification method is to use seed functions [13]. Seed functions are functions in the binary one wants to remove by removing the function and all functions that depend on it. These functions can specify a function in a specific binary, but cannot describe the same functionality over all binaries. An alternative is dual slicing [14] where a feature is defined by the difference in two program executions. So the program is started with two different parameters and the function calls that are present in only one trace are the functions of interest.

3.2 First Results

Using these building blocks we built a prototype demonstrating parts of our vision.

Identification. Our specification is given in the form of an implementation of the crypto functions we are looking for. We use these to find the functions of interest in the trace we recorded. The naïve approach of testing all functions called in the trace brute-force does not scale. We therefore employ domain-knowledge to narrow the search: A candidate function can often be identified by its use of certain constants (SBox in AES), specific CPU instructions (AES-NI), or heuristically by a density of bit-level operations in the code. To test the latter we use a machine-learning approach that is able to identify functions containing cryptographic operations with high confidence. A different approach is implemented in CryptoHunt [15] and the Software Analysis Workbench [16], where the authors translate the binary program into logical formulas that can be compared to a given reference implementation with an SMT solver.

Function identification is further complicated because we also need the order of parameters of the function in order to replace or invoke it. Again a brute-force attempt would be very slow, so we again use domain knowledge: The parameters we are interested in (plaintext, key, ciphertext) are pointers to memory buffers of at least 16 byte length. That significantly reduces the number of parameters we need to test.

Testing Cryptographic Implementations. Knowing the exact interface we can test the cryptographic implementation in the binary. We support two test modes:

Firstly, we support supervised encryption, which means that we check after every invocation of the encryption routine if the returned result is correct. This is done by running in parallel a trusted implementation. At this point we could also replace the entire crypto routine with the trusted implementation. Currently this check is done at runtime using DynamoRIO.

The second test mode is to run the encryption function against a list of well-known input-output pairs. In case of AES such pairs are provided by the NIST [17]. This works during runtime by waiting until the encryption function is first invoked and then repeatedly invoking just the encryption function with the chosen inputs and comparing the outputs to the specified ones. Any deviation is an indication that the encryption is not implemented correctly. This is essentially a from of differential testing [18].

Symbolic Analysis. We already use symbolic analysis to find so-called logic bombs in binaries. A logic bomb is a malicious action hidden in the binary and triggered on certain conditions. In terms of crypto this could for example be a backdoor leaking the key. A first attempt working on source code was already published [19] and we are currently busy porting this to the machine code level.

Replacing the Encryption Algorithm. The identification of the encryption algorithm and its parameters is the first important step to allow replacing the encryption algorithm. Our framework could be trivially extended to make the

replacement at runtime with DynamoRIO. This is because we already intercept the call to crypto functions for testing. Instead of running both functions and comparing the result, one could simply return the result of the reference implementation and never invoke the original function. While the runtime manipulation of the binary is perfect for testing, it is not desirable as a permanent solution due to the overhead. Therefore we need the binary rewriting tools outlined in Sect. 3.1 to persist changes in the binary itself.

Inserting Encryption. To insert encryption we need to specify insertion points, e.g. function calls to libc. libc is a standard C library used by virtually every Linux application. We can replace these calls with a transparent wrapper to encrypt data before it is passed to libc and decrypting data returned by libc. This could be done when writing data to a file or to a network socket. Of course this would be further complicated by adding key management and exchange. To protect the keys in memory from the original application we can use novel CPU-backed technologies that isolate certain parts of memory, such as Intel SGX.

3.3 Evaluation

Our current prototype implements the complete testing toolchain for AES: It has the ability to record traces, find the addresses and parameters of the crypto functions, and test the crypto function using NIST vectors. We implemented several models for AES using two modes (ECB and CTR) and various keylengths.

We can not only process the small toy examples we created for numerous ways to implement AES, but also the aescrypt2 sample program from the mbedTLS library [20]. All these example were compiled with GCC optimization level 3 and without symbols. The aescrypt2 example is 180 kb in size and contains more than 7000 assembler instructions.

4 Conclusion

We have presented our vision on how to address the challenges posed by cryptography in the life-cycle of executable programs. In order to automate the process of testing and adapting executables as much as possible, we propose to build a platform capable of analyzing, replacing, and inserting cryptographic functions with the goal of achieving a high level of automation. For this, we rely on a mix of techniques known from binary analysis and rewriting, program verification, model-based testing, and compiler construction.

Our envisioned platform will help analysts find flawed cryptographic implementations and replace them by trusted ones or even insert encryption functionality into executable programs. We have made promising first steps towards our vision by implementing parts of the platform on the Linux-x86-64 platform and applying it to different applications relying on AES. Our lessons learnt led us to new approaches for speeding up solving the identification problem of functions

and parameters, which we are implementing right now. We are also working on improving the symbolic analysis to make it more scalable and applicable to larger executables. Finally, we want to use rewriting techniques to make permanent changes to binaries as the next step in going after our vision.

References

1. Thompson, K.: Reflections on trusting trust. Commun. ACM **27**(8), 761–763 (1984)
2. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An empirical study of cryptographic misuse in Android applications. In: CCS 2013, pp. 73–84 (2013)
3. CCC: Analyse einer Regierungs-Malware. Technical report, Chaos Computer Club (2011)
4. Codenomicon, Google-Security: CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160, 3 Dec 2013
5. Bello, L.: CVE-2008-0166. Available from MITRE, CVE-ID CVE-2008-0166, 9 Jan 2008
6. Bruening, D., Zhao, Q., Amarasinghe, S.: Transparent dynamic instrumentation. ACM SIGPLAN Not. **47**(7), 133–144 (2012)
7. Hiser, J., Nguyen-Tuong, A., Hawkins, W., McGill, M., Co, M., Davidson, J.: Zipr++: exceptional binary rewriting. In: FEAST 2017, pp. 9–15 (2017)
8. Majlesi-Kupaei, A., Kim, D., Anand, K., ElWazeer, K., Barua, R.: RL-Bin, robust low-overhead binary rewriter. In: FEAST 2017, pp. 17–22 (2017)
9. Chipounov, V., Kuznetsov, V., Candea, G.: The S2E platform: design, implementation, and applications. TOCS **30**(1), 2 (2012)
10. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (state of) the art of war: offensive techniques in binary analysis. In: S&P 2016 (2016)
11. Saudel, F., Salwan, J.: Triton: a dynamic symbolic execution framework. In: Symposium sur la sécurité des Technologies de l'information et des Communications, SSTIC, France, Rennes, June 3–5 2015, SSTIC, pp. 31–54 (2015)
12. Cadar, C., Dunbar, D., Engler, D.R., et al.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI. vol. 8, pp. 209–224 (2008)
13. Jiang, Y., Zhang, C., Wu, D., Liu, P.: Feature-based software customization: preliminary analysis, formalization, and methods. In: HASE 2016, pp. 122–131 (2016)
14. Kim, D., Sumner, W.N., Zhang, X., Xu, D., Agrawal, H.: Reuse-oriented reverse engineering of functional components from x86 binaries. In: ICSE 2014, pp. 1128–1139 (2014)
15. Xu, D., Ming, J., Wu, D.: Cryptographic function detection in obfuscated binaries via bit-precise symbolic loop mapping. In: S&P 2017, pp. 921–937 (2017)
16. Dockins, R., Foltzer, A., Hendrix, J., Huffman, B., McNamee, D., Tomb, A.: Constructing semantic models of programs with the software analysis workbench. In: VSTTE 2016, pp. 56–72 (2016)
17. Bassham III, L.E.: The advanced encryption standard algorithm validation suite (AESAVS). NIST Information Technology Laboratory (2002)
18. McKeeman, W.M.: Differential testing for software. Digit. Techn. J. **10**(1), 100–107 (1998)
19. Papp, D., Buttyán, L., Ma, Z.: Towards semi-automated detection of trigger-based behavior for software security assurance. In: SAW 2018 (2018)
20. ARM: mbedTLS. https://tls.mbed.org/