Monitoring the Execution of Cryptographic Functions

Mathieu Amet, Oussama Ben Moussa, Guillaume Bonfante and Sébastien Duval

November 27, 2024

Monitoring the execution of cryptographic functions

Matthieu Amet[1], Oussama Ben Moussa[1,2], Guillaume Bonfante[1] and Sébastien Duval[1]

[1] Lorraine University - CNRS - LORIA, Nancy - France
[2] ENSEIRB-MATMECA, Talence - France

Abstract. We propose a new framework for the analysis of program execution, devoted to identifying cryptographic functions and retrieving cryptographic secrets. The need for a new tool arises from our experimental observation that the generic analysis tools are clearly too intrusive / resource-consuming for the inspected process, leading to failures such as timeouts. Thus our aim is to build dynamic monitoring tools as lightweight as possible to inspect the execution of sensitive code without impacting the execution.

In march 2024, the vulnerability CVE-2024-3094 targeting `XZ Utils` was revealed by the Microsoft software developper Andres Freund[3]. The backdoor was introduced within the library `liblzma` by a mysterious Jia Tan (their `github`'s name). In this attack, there were three challenges: the first one was to introduce the vulnerability within the targeted library, the second was to distribute it. The third one was to keep the backdoor discreet. Having a direct access to the source repository, Jian Tan could fulfill the two first ones in a single step. Unfortunately for him, but not for us, despite efforts, he failed at the last one. In which terms could we reproduce such an attack? Let us explicit our attack model. The attacker we have in mind knows that his victim is using some binary `EXE`. He knows that this `EXE` is manipulating some cryptographic functions. Finally, we suppose that the attacker will be able to patch some libraries on his victim's computer. On the other side, we do not suppose he has permanent access on the victim's machine, he has only the ability to patch a library. Furthermore, we make the hypothesis that the attacker has no access to the source of `EXE`. In other words, our challenge will be to forge the backdoor on a "black-box" executable. Finally, as a model of stealthiness, the patched library should behave as the original one up to the backdoor, and we only allow this library a constant time overhead.

We also slightly modify the attack goal: rather than obtaining a reverse-shell on the victim machine, our target will be to retrieve cryptographic secrets which allow, in a second step, to retrieve secrets or establish a remote connection to the victim machine.

From a more general perspective, the main goal of this article is to study the resistance of program execution to the recovery of cryptographic secrets.

[3] https://lwn.net/ml/oss-security/20240329155126.kjjfduxw2yrlxgzm@awork3.anarazel.de/

As a program runs, its instructions and registers can be recovered (for example using Intel's `Pin` functionality[4]), and it is only natural to wonder whether such knowledge can allow an attacker to recover some secret material that the program is using. Using `pintools` to perform dynamic analysis is not new. For instance, this is typically what we did in [2] where it was used to deal with self-modification. Deobfuscation is an other example, see for instance the nice work by Bardin, Potet and Salwan [16]. However, those preceding works require heavy resource-consumption.

We also study how much data is actually sufficient to recover secret information. Indeed, getting the full execution trace is slow, which (1) makes it easily detectable and (2) is impractical for real-time programs such as Internet connections (which timeout). Hence if we can identify parts of the trace which are sufficient for secret-recovery, this can make the approach stealthier and adapted to real-time applications.

Our main focus will be on the OpenSSL cryptographic library [18]. The reason to focus on OpenSSL is that it is largely used to build secure connections. It is the most commonly used library to implement the TLS protocol [14], which ensures the security in Internet connections (in particular in HTTPS connections). It is also the most commonly used library to generate cryptographic keys for use in remote-desktop communications (like SSH [19]) or file sharing (like GIT[5]).

Although OpenSSL is open-source, its code[6] is hard to read, as it contains layers of sub-libraries, layers of wrapping functions, and is highly customisable (as many cryptographic algorithms are available). Also, there is a gap between what is read in the source code and what is actually executed by a program, especially a program which is not OpenSSL directly but uses some OpenSSL functionalities. Therefore it is a good model for a blackbox analysis and we base our study on the program execution directly, rather than relying first on the source code and documentation.

Finding a secret within a binary is a quite common activity in reverse-engineering, think of "Capture The Flags" challenges. For that sake, symbolic analysis has shown to be very powerful. It is one of the use case of tools such as `Binsec`, `Angr` (see [17]) or `MIASM` (e.g. [5]). However, symbolic analysis takes a lot of computational power. Our approach relies only on an analysis of a trace. The closest work we are aware of is [6] where the authors use traces as a basis for symbolic analysis. Compared to their approach, on one side, we do not use symbolic analysis, on the other side we focus on a much more specific target, that is cryptographic functions analysis.

Paper structure. First, we design a tool which allows to identify the parts of an execution trace that manipulate cryptographic secrets (inside an Internet connection in our use-case). With this isolation of the cryptographic part of the

[4] `https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html`

[5] `https://git-scm.com/`

[6] `https://github.com/openssl/openssl`

trace, useless parts of the trace can be thrown away. In a second step, we analyse in detail a specific cryptographic execution, namely the execution of `openssl genrsa`, and we show that the private key material can be recovered during execution.

1 Identifying the cryptographic parts in the execution of of an Internet connection

In this section, we will study the execution traces of several Internet connections, with the aim to identify parts of the traces which execute sensitive operations, such as the cryptographic operations of the TLS protocol. In order to ensure that cryptography is present, we focus on HTTPS connections, and for ease of scripting, we use `curl`[7] to launch the connections.

1.1 Obtaining and filtering the execution traces

We wrote a little scraper with the aim to establish some connections via four different encryption protocols, namely:

– TLS_AES_256_GCM_SHA384
– TLS_AES_128_GCM_SHA256
– TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
– TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

These are the most common TLS configurations (we verified it experimentally on about 5000 websites).

They give a diversity of cryptographic building blocks: two AES versions [4], the GCM mode [15], two SHA2 variants [7], ECDHE [8] and RSA [10]. Furthermore, the variations on the keys length avoid patterns related to that factor.

Concerning the web connection, the scrapper uses `curl` without specific options. The cryptographic connection protocol is retrieved by means of `wireshark`[8]. After that step, we could select 50 websites for each of the four categories. For each of the selected websites, we run our two_byte tracer:

```
pin -t "two_bytes.so" -- curl website
```

One word about the `two_byte` tracer. For each instruction, it outputs the first two bytes of the instruction. Why not the instruction itself, or its address? The problem of the latter idea is that the amount of data is so huge that the traced program becomes too slow to keep a reliable connection to the target website. In that case, traces simply end on a "timeout". Keeping two bytes for each instruction is sufficient to identify them roughly. To conclude, generally speaking, we get traces having a size around 5GB, thus the limitation to a small number of websites.

[7] `https://curl.se/docs/manpage.html`
[8] `https://www.wireshark.org/`

1.2 Analysis of function calls

One way to reduce the trace size while keeping the structure is to keep only the function calls (ASM instruction `call`). We will use it to identify the parts of the execution which execute cryptographic functions.

The trace of function calls yields files of about 1 MB,

We start with several issues that arise, then make observations on the sensitive instructions that we identify.

Parallelism. Through this filtering, we identify an issue: a high level of parallelism, which makes the structure of the program hard to grasp. This complexity can be canceled if the attacker/analyser is the one launching the execution, in which case they can turn off the parallelism. In other cases, parallelism will render the analysis more complex. The pintool is able to separate traces between parallel threads to some extent, but we observe that this separation does not always work. In particular, calls to OpenSSL's `CRYPTO_THREAD_xxx` functions, which launch a new thread (either a pthread or a Windows thread depending on the operating system), do not seem to be separated by the pintool.

Black-box calls. We also observe that OpenSSL's `libcrypto` uses some functions in an opaque environment called "exdata". Everything in these functions is executed in a black box ensured by what OpenSSL calls "BIO". The idea is that for each of these calls to a BIO box, only the inputs and outputs can be seen, and not what is happening inside the box. That is the goal of BIO according to OpenSSL's documentation, and it seems to thwart the pintool well enough, at least with no further work.

Cryptographic function initialisations. Another issue for the analysis is that even if OpenSSL will only use few ciphers and hash functions during the TLS connection, it still initialises *all* ciphers (called `EVP_CIPHER`) and hash functions (called `EVP_DIGEST`) available in its library. This means that if we aim to identify which cryptographic functions are used in the connection by filtering instructions which contain their name, every cryptographic function will have at least a few function calls using its name during the initialisation. Identifying which cryptographic function is actually used is therefore slightly harder.

Elements on cryptography and TLS. At this point, we would want to identify which function calls in a TLS connection manipulate sensitive variables. This requires some understanding of how TLS works, hence we will give a short summary of the main steps and sensitive variables in a TLS connection from a theoretical point-of-view.

Let us first recall the three main cryptographic types of tools: 1. ciphers, which allow to make data readable only to those having a secret key, 2. signatures, which allow to verify the identity of the sender of some data and 3. cryptographic hash functions, which allow to verify the integrity of the data (*i.e.* that is was not modified).

Ciphers manipulate secret keys during the decryption of data (not necessarily during its encryption). Signatures manipulate secret keys during the server signing operation (not when the client verifies the signature). Hash functions manipulate no secret key.

TLS (for Transport Layer Security) is a protocol that can be plugged onto any communication protocol to ensure the security of the connection. It works in several steps, and some important steps happen before the connection.

1. Key generation (before connection): the server creates a couple (`public key`, `private key`). These variables allow anyone knowing the `public key` to securely connect to the server. The security of this connection requires the `private key` to remain secret, hence `private key` is sensitive.
2. Public key sharing (before connection): the server must then share its `public key` with the client. Although this key is not confidential, its integrity and the authentication of its owner are required, which requires a secure means of communication. This step is complicated and does not rely directly on cryptography (or on secrets). It relies on Public-Key Infrastructures (PKI), usually based on a public-key certificate (a tuple containing the `public key` and the server URL, plus other details).
3. Shared-secret generation (connection start): for efficiency reasons, a secure cryptographic channel relies on symmetric-key cryptographic algorithms, which require the client and server to know a common secret. Creating this common secret is the goal of this crucial step. In TLS, the `shared secret` generation is handled through a variant of the classical Diffie-Hellman algorithm (precisely Elliptic-Curve Diffie-Hellman Ephemeral). This secret sharing requires that both client and server share the `public key`. The `shared secret` is sensitive.
4. Secure session (during connection): using the `shared secret`, server and client can communicate securely. Several algorithms can be used to this end, but in TLS (and in general) most of these algorithms rely on a cryptographic block-cipher called `AES` (in TLS, the variant is usually `AES-GCM`).

With these theoretical elements in mind, let us see where secrets are manipulated in practice.

Public-key certificates. Parallelism and cryptographic function initialisation add some amount of noise to the traces. Another source of noise comes from the public-key certificates.

Let us remind how public-key certificates work [1]: in order to start a secure connection, the server and the client need to share a secret key (which will be used by symmetric-key algorithms, such as `AES-GCM`). This secret key is shared using a public-key algorithm (usually ECDHE). ECDHE relies on the server's pair of keys: a private key and a public key. For ECDHE to work, the server's public key must be sent to the client. Yet although this public key is not confidential (by definition), it must still be sent securely to ensure its integrity and the identity of its owner (using a *signature*). This secure sending of the public key is done using a Public-Key Infrastructure, which shares the public key in a public-key certificate. This certificate contains various information, including (1) the server's URL, (2) the server's public key and (3) the server's public key, signed *by a third party* (a Certificate Authority).

This is important to understand, because this means that the client must use a signature algorithm to verify the certificate. And contrarily to every other cryptographic function, this signature algorithm *is not chosen during the TLS connection*, as it was chosen prior to the connection by the certificate authority.

This means that in every trace, for every public-key certificate on which the connection depends, there will be a call to a signature algorithm that we cannot predict. In every trace, we actually observe that many certificates are verified, for two reasons. First, because the certificates actually depend on a hierarchy of Certificate Authorities, not just one, and all their certificates need to be verified. Second, because most web pages actually contain plug-ins from other websites, which require a connection to each of the websites and a verification of their certificate.

Overall, many signature algorithms are called during every trace, and we cannot predict which. It is important to understand that these signature calls are **noise**! They do not manipulate any secret, as they are only signature *verifications* (they only use public keys).

Sensitive instructions. To understand which instructions may manipulate cryptographic secrets, some understanding of cryptography is required. For instance, in the previous paragraph, it is clear that the signature verifications cannot manipulate any secret values.

Other cryptographic functions do not manipulate any keys. This is the case of cryptographic hash functions (such as `SHA1`, `SHA256`, `SHA384`, `SHA512` and `SHA3` [7]). One could argue that such hash functions are sometimes used to hash sensitive data, as is the case when combined with RSA [10] to sign messages, but this does not happen in a TLS connection. The only case in which cryptographic hash functions do manipulate sensitive data in a TLS connection is in the case of HMAC [11], a Message Authentication Code algorithm that uses hash functions. In that case, recovering the input of the hash function may yield information that could endanger the integrity of the connection. However nowadays, HMAC is only supported by very few websites, and is never the default algorithm.

We are sure that some cryptographic functions do manipulate secrets: it is the case for AES [4], which is the block-cipher used in almost every TLS connection. Hence we look for instructions concerning AES. In the traces, AES is called in 4 zones of instructions: (1) 2 calls in the initialisation, (2) a few calls in a zone of 200 function calls, (3) a few calls in a zone of 7000 function calls in the middle of the trace and (4) a few calls in a zone of 120000 function calls towards the end of the connection. All these calls are actually to the instruction `AES_unwrap_key`. If other function calls are related to AES, they would be in the BIO black boxes.

Overall during the core of a TLS connection, the only cryptographic secrets that are manipulated are the symmetric keys, and they are almost always used by AES, and we can positively identify the zones in which AES is executed by the program.

1.3 Automatic identification of the zones which manipulate sensitive keys

In order to identify the zones of instructions that manipulate sensitive keys, we will focus on AES based on the previous analysis. We will detect calls to `AES_unwrap_key`.

Of course, if the attacker/analyser has access to the name of function calls, they can directly detect calls to `AES_unwrap_key`. We will now focus on a less powerful attacker/analyser who does not have the names of functions, and we will see if such an attacker can still identify calls to `AES_unwrap_key` during the execution.

Note that for the rest of Section 1.3, we will consider all instructions of the traces, not only ASM instruction `call`.

Learning. The idea will be to learn the execution pattern of `AES_unwrap_key` using supervised learning, then use this to detect the execution of `AES_unwrap_key` during a full trace execution.

For the learning, we will use traces of `AES_unwrap_key` as examples of *positive detection*, and execution traces not containing `AES_unwrap_key` as examples of *negative detection*.

Final format of the data. To identify the execution pattern, we do not require the whole instruction trace. We will only keep the first two bytes of each instruction (the opcode), which identifies the type of operation that is applied. This allows to largely reduce the trace size to around 500 MB.

Learning data. The data is the full execution trace of an Internet connection. Within it, we will look for windows of 50 instructions corresponding to `AES_unwrap_key`. To allow for real-time detection during an execution, we will cut the trace into blocks of 200 instructions, and we will perform the detection of the 50-instruction window inside of this 200-instruction block.

For positive learning traces, we position the 200-instruction block around a call to `AES_unwrap_key`, making sure to capture at least the 50-instruction window after the call to `AES_unwrap_key`.

For negative learning traces, we pick a random position and extract a 200-instruction block. We check that there is no call to `AES_unwrap_key` inside of this block, otherwise we pick another one.

In proportion, we generate about 10 times more negative cases than positive ones, as learning the negative case is harder (it is more diverse).

In total, we use 18202 negative traces and 1841 positive traces for the learning phase, then 513 negative traces and 509 positive traces for tests.

Learning tools and results. We used several learning tools, as it was not obvious which method would be the best:

– Recurrent Neural Network [13], and more specifically Long-Short Term Memory [9], which is a deep learning method fit for the detection of sequences, based on time relation between data. Our RNN was made of an embedding part for preprocessing of traces (extraction of trace characteristics), then the learning LSTM part made out of 64 gates, and finally a dense part to classify and extract the result as a 0 or 1. The results of using the RNN are summarised in the following confusion matrix:

$$\begin{bmatrix} \text{True Positive} & \text{True Negative} \\ \text{False Negative} & \text{False Positive} \end{bmatrix} = \begin{bmatrix} 357 & 9 \\ 152 & 504 \end{bmatrix}$$

  Our RNN is excellent at distinguishing negative cases, but is less precise for positive cases. We compute a score of accuracy of 0.784 (as the mean between true positives and false negatives, weighted by number of positives and number of negatives).
– Convolutive Neural Network [12], which is a deep learning method fit for the detection of patterns. Our CNN was made of a convolution part for preprocessing of traces, then a max pooling to extract the most saliant characteristic, then another convolution and another max pooling steps, followed by a flatten part to extract the data as a 1-dimensional array, and finally a dense part to classify and extract the result as a 0 or 1.
  To represent our data as a picture, we transformed each instruction into a pixel, where the first opcode byte is used for Red, the second opcode byte is used for Green, and the Blue is set to 0. The results of using the CNN are summarised in the following confusion matrix:

$$\begin{bmatrix} \text{True Positive} & \text{True Negative} \\ \text{False Negative} & \text{False Positive} \end{bmatrix} = \begin{bmatrix} 373 & 29 \\ 136 & 484 \end{bmatrix}$$

  The results with the CNN are similar to the RNN case, with a small improvement for the detection of positive cases. The accuracy score is 0.8053, slightly better than the RNN.
– Support Vector Machine [3], which is fit for classification and regression. We kept the default linear separation of the SVM and got the following results:

$$\begin{bmatrix} \text{True Positive} & \text{True Negative} \\ \text{False Negative} & \text{False Positive} \end{bmatrix} = \begin{bmatrix} 485 & 2 \\ 24 & 511 \end{bmatrix}$$

  The SVM is better than the CNN and RNN at distinguishing the negative cases, and is also very good at distinguishing the positive cases. This gives an accuracy score of 0.9907. The SVM model therefore seems quite fit for learning in our case.

Conclusion. Overall, using an SVM, we manage to successfuly learn when and where cryptography is used in an execution trace (and more precisely in this case of the `AES_unwrap_key` function), and validate that the knowledge of the function calls is not required, the opcode of each instruction is sufficient.

The size of the window (200), the number of instruction (50) do not look to be crucial factors. We got almost the same results when shifting them slightly. For instance, for window size, we even tried windows of 1k.

This can be used for a monitoring program, which recovers the trace in real time, and can use this detection to throw away the trace parts which are useless for secret recovery. In a second part, we will study how to recover secrets in the cryptographic parts of a trace, now that we have a way to isolate this part.

2 Analysis of OpenSSL's RSA key generation

In the first part, we showed in which terms, we could perform some coarse grain analysis of cryptographic libraries, by coarse grain, we mean at function level. In this second part, we want to perform a finer inspection of the binary. We would like to extract precisely the instructions that deal with cryptographic data. Of course, we could do it via reverse engineering. But, let us try to do it in a completely automatic way.

We consider the following attacking scenario. The attacker would like to perform some identity fraud with respect to some victim. For that sake, they have the possibility to replace one library by an other one on the victim's machine. Then, identifying the instructions manipulating keys makes sense. Indeed, suppose you can determine that for the program used by the victim, the instruction at relative address `a` within library `L.dll` deals with the secret key, say in register `RAX`. It is fairly easy to transform the library so that it retransmits the contents of the registers `RAX` via an auxiliary channel to the attacker. You "simply" have to hook the instruction at address `a` to the payload and then return back to normal operations.

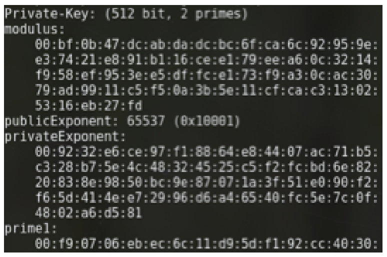
Now that our plan is set, let us do it on a concrete example. Again, we will focus on an example of program which manipulates cryptographic secrets: the generation of a private/public RSA key pair using OpenSSL's `genrsa` command. This key generation is classically used to generate cryptographic keys used in protocols like GIT or SSH, and is similar to the key generation of TLS.

All along, we consider the execution of `openssl`:

```
openssl genrsa -out {file} [512|1024|2048]
```

that will create a file containing the newly generated key. Then, applying

```
openssl rsa -in {file} -noout -text
```

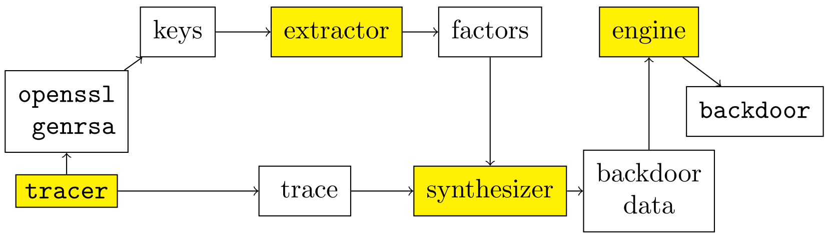


```
Private-Key: (512 bit, 2 primes)
modulus:
    00:bf:0b:47:dc:ab:da:dc:bc:6f:ca:6c:92:95:9e:
    e3:74:21:e8:91:b1:16:ce:e1:79:ee:a6:0c:32:14:
    f9:58:ef:95:3e:e5:df:fc:e1:73:f9:a3:0c:ac:30:
    79:ad:99:11:c5:f5:0a:3b:5e:11:cf:ca:c3:13:02:
    53:16:eb:27:fd
publicExponent: 65537 (0x10001)
privateExponent:
    00:92:32:e6:ce:97:f1:88:64:e8:44:07:ac:71:b5:
    c3:28:b7:5e:4c:48:32:45:25:c5:f2:fc:bd:6e:82:
    20:83:8e:98:50:bc:9e:87:07:1a:3f:51:e0:90:f2:
    f6:5d:41:4e:e7:29:96:d6:a4:65:40:fc:5e:7c:0f:
    48:02:a6:d5:81
prime1:
    00:f9:07:06:eb:ec:6c:11:d9:5d:f1:92:cc:40:30:
```

the details of the cryptographic keys are printed by OpenSSL (into the standard output). The secrets are named `privateExponent`, `prime1` and `prime2`. But can we retrieve those data at generation time, not *a posteriori* as we did here. We will focus on the determination of the private exponent, but that is more or less arbitrary and actually, we will have access to all parameters[9]. Naturally, as stated in introduction, there is a surreptitious usage of this idea. Such a tool could serve as a hidden door within an attack as we described in introduction. In other words, outside protocol analysis, achieving our goal can be interpreted as a security issue.

Final preliminary remark: we have made experiments on `openssl`'s key generation, but our approach is more generic. We only rely on two hypotheses: 1. that the program uses some key, hidden or not, and 2. that this key could be retrieved by a legitimate user after program execution. This second hypothesis is somewhat limiting, but it is necessary to generate learning data for our tool.

Our general plan is drawn on the figure below with time flying from left to right:



Roughly speaking, our learning process works as follows. We run the executable `openssl` within a $\boxed{\text{tracer}}$. At the same time, or just after the execution, we extract the "secret": this is the box $\boxed{\text{extractor}}$ in our schema. In the present case, we will take the private exponent within the private key. Finally in the $\boxed{\text{synthesizer}}$ phase, from this knowledge of the secret and the execution trace, we build backdoor data, that is instructions, registers or memory that specifically manipulates the secret, obtained by correlating the trace with the known private exponent ("factors"). In other words, the synthesier returns the main ingredients to build the backdoor, that is done by the $\boxed{\text{engine}}$. The backdoor

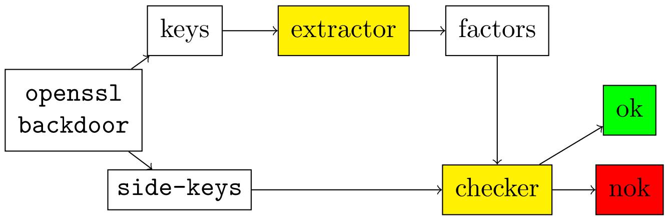
[9] Also in an RSA key, the knowledge of the private exponent or one of the primes is enough to recover the full key.

behaves exactly as the original program `openssl` up to the exception that it extracts the private exponent in a separate channel.

Since we do not aim to create real backdoors, actually, our engine forges a pintool `openssl_backdoor.so` such that running the pintool behaves exactly as expected: it will perform as `openssl` does, except that for RSA key generation, it will save the secret key in a side file. In short, we replace `openssl args` by `pin -t openssl_backdoor.so args`.

One could argue that saving the key in a local file is not really useful from an attacking perspective. However, transforming that to a remote connection is standard engineering. This could be done in an additionnal step. We rather focus on the data stored within this file, called side-keys in the following.

The last step is to validate that the side-keys correspond to the real secret. On our testbed, we check that the backdoor pintool provides the same data our extractor does. The `openssl backdoor` outputs keys (normal behavior) together with data in a side channel (side-keys). We extract the secret factors as we did above with $\boxed{\text{extractor}}$, and we $\boxed{\text{check}}$ that they agree with the side-channel values.



In this process, the tracer and synthesizer are generic, while the extractor is specific to the target program and the searched secret.

2.1 Obtaining and filtering the traces

The tracer is in charge of extracting an execution trace from a run. Our tool records instructions' address and data values at every point during execution. For a program, that may lead to huge amounts of data. Typically, a run has a length of magnitude 10^9 instructions. Recording addresses (8 bytes long), of 10 registers (each being 8 bytes long) leads to a file of approximately 100GB. Thanks to Section 1, we can parametrize our tool to restrict data recording to some specific libraries (for instance `libcrypto`) and even more to specific functions.

Furthemore, we can change the list of considered registers. These two tricks save a lot of space in practice. For our experiments, we got trace files with size of the order of 1GB. We managed those hyper-parameters manually, but it is clear that we could do it via an automatic process.

By data recording, we mean two sorts of entries. First, we may record the value of some register, say `RAX`, `XMM2` and so on. But that is not sufficient in practice. We also record pieces of memory corresponding to the ”Memory Effective Address” denoted by the instructions. Registers come with an underlying size. For memory, we use a parameter. So far, a size of 32-bytes has shown to be sufficient. To simplify the management of all these parameters, we use a global configuration file.

Trace content. In the end, after application of our `pintool`, we get a (binary) file storing each instruction's address and a list of values (with their underlying size):

```
struct entry{
    uint64_t address;
    uint8_t * value;
    uint8_t size;
}
```

where `values` denotes the concatenation of recorded data. Given an entry `E`, we denote by `E.address` its underlying address and by `E[r]` the value corresponding to the data `r`. Recall that `r` denotes either a register (hence the notation) or the memory effective address.

2.2 Recovering the private RSA keys

The analyzer is devoted to a post-treatment that retrieves key information. For instance, for the command `openssl genrsa` mentioned above, we get back the private exponent corresponding to the key with the command

```
openssl rsa -in new_key_file.key -noout -text
```

followed by a simple filtering process. The result is stored within a file that will be read by the synthesizer.

2.3 Trace analysis

The learning phase of the synthesizer works as follows: we take the true value of the secret extracted by the `extractor` tool and the execution trace in the form of entries in file `binary_trace.bin`. The secret is a sequence of bytes next denoted `k`, of length `len(k)`, while the entries of the trace are denoted as S and their size as N. Recorded registers are denoted `R`.

In a first step, we need to filter instructions/registers that share some part of the secret. For this purpose, we build the following dictionary `D`. Its keys are pairs (i, r) with $i \leq N$ and $r \in$ `R`. To any key, we associate the longest substring between the registers value and the secret. Actually, since factors of length 1 and 2 are way too common, we restrict the dictionary to sequences of size at least 4.

In a second step, we consider the set of instructions occurring within the trace. Instructions are identified by their address `a = S[i].address`. Let an address `a`, we compute the (ordered) list

```
S_a = [ i < N |  a = S[i].address]
```

In other words, this is the list of entries specific to some address a.

An address a and a register r are said to be compatible with the key k whenever there are three constants α, β, γ and an increasing list of indices $\ell_0, \ell_1, \ldots, \ell_{\texttt{len(k)}} \leq \texttt{len}(\texttt{k})$ such that for all $0 \leq j < \gamma$,

- $\ell_0 = 0$, $\ell_\gamma = \texttt{len}(\texttt{k})$
- $\texttt{D}[\texttt{S}_\texttt{a}[\alpha + j \times \beta], r] = [k_{\ell_j}, \ldots k_{\ell_{j+1}}]$,

In other words, after α applications of the instruction at address a, every β application of this instruction, the content register r at position q will perform a linear sweep within the secret. We afford to drop the last γ instructions within the trace. Sometimes, we observed that two loops are necessary to get a correct adequation between the register and the key. Hence the factor β.

To sum up, identifying a covering sequence means we are capable of rebuilding the key from the value of registers at corresponding instructions.

2.4 Pintool synthesis

We suppose we succeeded to find a sequence $\{(a_1, r), \ldots, (a_t, r)\}$ as above with the respective parameters. Then, we are capable to build a pintool that will skip the first occurrences of the applications of an instruction. Dropping the last instructions can be done via a buffer. Finally counting modulo β is (almost) immediate.

So, actually, the main difficulty of this part is to choose the "best" sequence. Indeed, once there is one, we noticed that there are actually many. Some will involve more instructions, leading to a constant time factor efficiency loss.

Since the pintool is based on interruption on a very small number of instructions (usually, even 1, when the succeeding sequence is composed of a singleton), the time overhead is marginal.

2.5 Result

Concerning out dataset, we tested our tool on the `Windows` platform, on different Linux platforms (Ubuntu, Debian) with the tool as presented above. All the experiments we did so far succeeded : the addresses, the registers we found contain the secret data.

Further, again, to test the robustness of the approach, we tested different versions of the libraries. We made the experiment on `libcrypto` 3.0, 3.1 and (3.2) (and some subvariants). We got positive results in all cases.

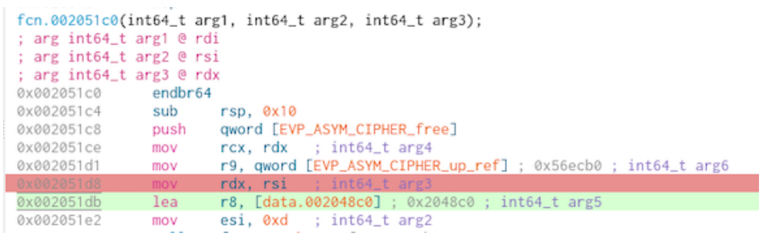
Even more, we tested the tool on a `Raspberry Pi.3` to test if an other processor (here, an ARM-32bits little-endian) would lead to an other conclusion. For that sake, we had to use an other tracer since `pin` only works on x86 platforms. We transposed all the work on the `ARM` processor, thanks to the `dynamorio`[10] framework. All the conclusions remain the same : we find the secret.

[10] `https://dynamorio.org/`

Finally, we tested the tool on different key lengthes, 512, 1024, 2048, 3096. First, in each case, we get a correct answer. But second, all the results are compatible. The addresses we got for a key length of 512 are those for the other sizes. From the attacker perspective, this is good news. He will not have to steal the information before the attack.

2.6 Discussion

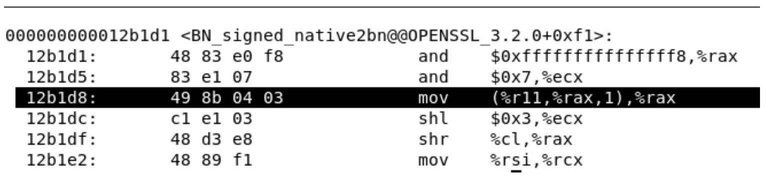
First, in each cases mentionned above, the addresses that the tool provides were lying within the `libcrypto` library. Here is one of the example displayed with `Cutters`:



```
fcn.002051c0(int64_t arg1, int64_t arg2, int64_t arg3);
; arg int64_t arg1 @ rdi
; arg int64_t arg2 @ rsi
; arg int64_t arg3 @ rdx
0x002051c0      endbr64
0x002051c4      sub     rsp, 0x10
0x002051c8      push    qword [EVP_ASYM_CIPHER_free]
0x002051ce      mov     rcx, rdx    ; int64_t arg4
0x002051d1      mov     r9, qword [EVP_ASYM_CIPHER_up_ref] ; 0x56ecb0 ; int64_t arg6
0x002051d8      mov     rdx, rsi    ; int64_t arg3
0x002051db      lea     r8, [data.002048c0] ; 0x2048c0 ; int64_t arg5
0x002051e2      mov     esi, 0xd    ; int64_t arg2
```

In the latter example, the tool is able to name some of the structures involved in the computation. We recognize some zone in memory starting with the prefix `EVP_ASYM_CIPHER`, some central data of the cryptographic mechanisms. When we made the snapshot, we knew that we were on good tracks.

Second, we encountered a large variety of registers: `rax, rsi, ymm0, r11` and even in memory. Here an example of the tool on the ARM processor:



```
000000000012b1d1 <BN_signed_native2bn@@OPENSSL_3.2.0+0xf1>:
  12b1d1:       48 83 e0 f8             and    $0xfffffffffffffff8,%rax
  12b1d5:       83 e1 07                and    $0x7,%ecx
  12b1d8:       49 8b 04 03             mov    (%r11,%rax,1),%rax
  12b1dc:       c1 e1 03                shl    $0x3,%ecx
  12b1df:       48 d3 e8                shr    %cl,%rax
  12b1e2:       48 89 f1                mov    %rsi,%rcx
```

Final point of discussion. All experiments were done with a time limit of 1 minute. Beyond, we think that we enter the domain of symbolic analysis. We used standard (standalone) machines, for instance an Intel i5-1140G7. Building the trace takes around 20 seconds (to be compared with around 2 seconds without trace recording). The number of recorded data is the main factor for this step of the analysis. Then, the synthesis takes also around 30 seconds (again, depending on the size of the data).

3 Conclusion

In this work, we have studied the recovery of secrets during program execution. It appears that during the execution of a program using some OpenSSL functions, it is possible to identify where and when cryptographic functions are executed. This allows to isolate the sensitive parts of the execution. Identifying these parts does not require much information, only the opcode of the instructions, and can be done on-the-fly during an Internet connection.

In a second part, we have demonstrated how one can recover some secret value during the execution of an OpenSSL function. We experimented on `openssl genrsa`, but our tool is hardly limited to this use-case. We positively recover the private key of the RSA key generation during the execution of `genrsa`.

Our tool can be downloaded on our website.

Future works. Several options arise for future works. Much of OpenSSL's sensitive computation appear to be happening inside of black boxes. These are meant to resist pintool analysis, but we did not put much effort to penetrate these black boxes in this work. Sutdying their actual resistance thus seems like an important work.

Another interesting direction is to keep on evaluating how much data is actually required to recover cryptographic secrets. It appears that the execution trace is already very powerful, but also that the opcodes only are enough to recover a lot of information. An attacker/analyser does not always have a strong access to the user's account or to the fully-detailed instructions trace. However some function of the instructions is more available, such as side-channel traces, whether they be time consumption, energy consumption or electromagnetic emissions. These leak information on the program during execution. Although this information is not full and noisy, it is available to anyone with a physical access to the device. A typical model of electric leakage is to assume that the attacker/analyser can recover the Hamming weight of each register (that is the number of bits to 1 in the register). It would be interesting to see if it is still possible to recover the secrets while relaxing the needs of the attacker/analyser.

Acknowledgement This work has been partially supported by the French Agence Nationale de la Recherche through the France 2030 program under grant agreements No. ANR-22-PECY-0010 CRYPTANALYSE and ANR-22-PECY-0007 DEFMAL.

The authors would like to thank `CyberHumanumEst`[11], a cyber-war that we organize at the Université de Lorraine. This annual event serves as a support for our attack scenarios.

All experiments described here were carried out in the High Security Laboratory [12] of the LORIA. Thanks to Fabrice Sabatier for his help in running the platform.

[11] https://cyberhumanumest.org
[12] https://lhs.loria.fr

References

1. Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and David Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.
2. Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *22nd ACM Conference on Computer and Communications Security*, Denver, United States, October 2015.
3. Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
4. Joan Daemen and Vincent Rijmen. *The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition*. Information Security and Cryptography. Springer, 2020.
5. Bruce Dang, Alexandre Gazet, Elias Bachaalany, and Sbastien Josse. *Practical Reverse Engineering: x86, x64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. CreateSpace Independent Publishing Platform, North Charleston, SC, USA, 2014.
6. Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se: A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656, 2016.
7. PUB FIPS. 180-2. *FIPS Publication–Secure hash standard (+ Change Notice to include SHA-224)*, 2002.
8. Ibrahim Hajjeh and Mohamad Badra. ECDHE_PSK Cipher Suites for Transport Layer Security (TLS). RFC 5489, March 2009.
9. Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
10. Jakob Jonsson and Burt Kaliski. Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. RFC 3447, February 2003.
11. Dr. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, February 1997.
12. Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.
13. Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5:115–133, 1943.
14. Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.
15. Joseph A. Salowey, David McGrew, and Abhijit Choudhury. AES Galois Counter Mode (GCM) Cipher Suites for TLS. RFC 5288, August 2008.
16. Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From virtualized code back to the original. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, volume 10885 of *Lecture Notes in Computer Science*, pages 372–392. Springer, 2018.
17. A. Tang. Solving for Hashes in Flare-On, 2015.
18. The OpenSSL Project. OpenSSL: The open source toolkit for SSL/TLS. `www.openssl.org`, April 2003.
19. T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254 (Proposed Standard), January 2006.