Make Crypto Safe Again!

Detecting Bugs in API Usage Using Bounded Model Checking

Matheus V. X. Ferreira Malte Möser

1 Introduction

The codebases of many popular cryptographic libraries are the result of decades of development and incremental changes. OpenSSL for example, arguably the most important crypto library available today, has been around since 1998 [6]. As a result, the libraries' application programming interfaces (APIs) have been continuously adapted and changed, often increasing their complexity and making them harder to use correctly.

Correct API usage is especially important in the context of information security, because neglecting even minor details can lead to a complete break of the security the protocol is intended to provide (cf. [2]). With OpenSSL, mistakes such as ignoring a single return statement have been observed to be able to lead to a complete breach in confidentiality [3].

A recent line of academic work has applied model checking techniques to discover API misuse (cf. [3, 8]). In this context, the goal of this project is to evaluate the feasibility of using Bounded Model Checking techniques (in particular, the Bounded Model Checker CBMC [1]) to validate correct API usage. For this, we will develop property monitors that allow to check that programs follow a correct sequence of API calls.

The remainder of this report is structured as follows. Section 2 presents the necessary background on the validation of SSL certificates. Section 3 explains how we can use property monitors in combination with a Bounded Model Checker such as CBMC to validate correct API usage, and Section 4 shows the concrete validation we perform for OpenSSL. Section 5 presents some examples of applying our property models and discusses the limitations of this approach. Finally, Section 6 concludes the report.

2 SSL Certificate Validation

Transport Layer Security (TLS) and its precursor Secure Socket Layer (SSL) are protocols enabling secure communication on the Internet. They are designed to resist active and passive attacks and to achieve the following three protection goals. First, end-to-end encryption between client and server ensures confidentiality of the communication contents. Second, message authentication ensures integrity of the communication contents, which means that modifications of the content can be detected. Third, a public-key infrastructure allows to authenticate the communication partners to prevent a malicious man-in-the-middle attacker from impersonating the counterparty. In modern application, be it reading

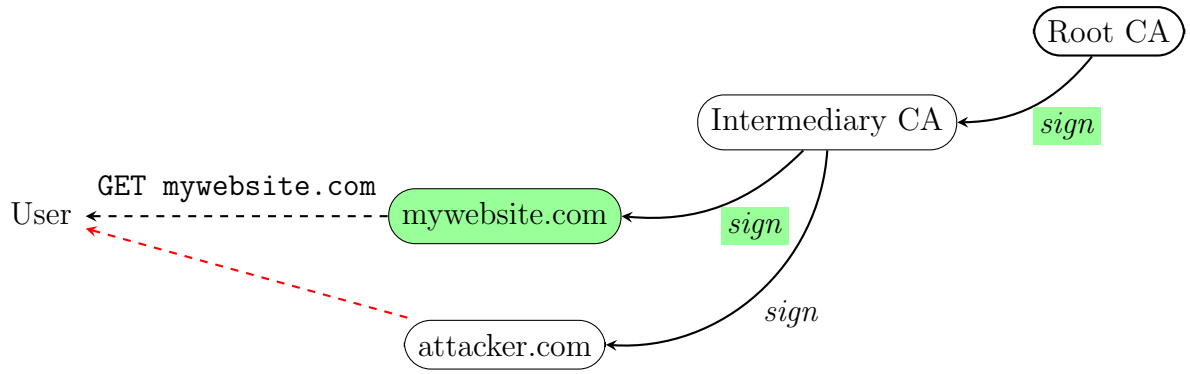

Fig. 1: Chain-of-trust and hostname verification (the user needs to verify the hostname and the signature highlighted in green)

your emails on Gmail or uploading pictures to Instagram, the end user typically initializes a connection to a server and then validates its authenticity (but not the other way round).

In this project we focus on the challenge of SSL certificate validation using the popular library OpenSSL. A certificate basically represents an attestation that a public key belongs to a certain entity, such as a hostname. Certificate validation is essential to prevent a man-in-the-middle attacker from being able to intercept and manipulate the communication. The validation of certificates consists of multiple steps (cf. [3, 2]):

Chain-of-trust verification. To validate the authenticity of a certificate we need to validate its chain-of-trust. SSL certificates are issued by certificate authorities (CA), an intermediary that must be trusted to only issue certificates to the owner of a certain domain name. CAs are organized hierarchally, i.e. a root CA (whose certificate is usually available on the user's machine) issues a certificate for an intermediary CA, and the intermediary CA issues a certificate for a domain name (cf. Figure 1). For each step in this hierarchy we need to verify that the certificates have been correctly signed.

Hostname verification. A valid certificate must belong to the hostname that a user tries to connect to. Otherwise, an attacker could intercept the communication using a certificate that is valid but belongs to a different domain name.

Certificate revocation When the private key of a server has been lost or was stolen, the corresponding certificate should be invalidated. A client thus needs to check whether a particular certificate has been revoked.

In practice, these steps have proven to be challenging due to the complicated APIs of cryptographic libraries [2]. For example, OpenSSL will validate the certificate chain but the developer must manually request and evaluate the verification result. Even more, the validation will return a success if no certificate has been presented at all [3]. While OpenSSL *can* also be configured to automatically abort the connection if the certificate or the chain-of-trust is invalid, this is not a default.

Verifying the hostname differs greatly between versions of OpenSSL [5]. Version 1.0.1 and earlier did not provide any hostname checks, the developer needs to implement such

a check herself. Version 1.0.2 provides a set of configuration parameters to enable built-in hostname verification, and Version 1.1.0 (finally) automatically validates the hostname.

Certificate revocation cannot be easily checked using OpenSSL APIs. Recent work by Liu et al. [4] furthermore demonstrates that certificate revocation is generally an unsolved problem. Only a tiny fraction of certificate revocations are available through existing certificate revocation infrastructure, and most client software does not bother checking these revocation lists at all.

3 Code Validation using CBMC and Property Monitors

CBMC [1] is a software tool that performs Bounded Model Checking of C and C++ code. While it is able to check a wide variety of aspects of the code, such as buffer overflows and pointer safety, we will mostly make use of its ability to verify user-specified assertions in the context of property monitors. A property monitor allows us to track changes in internal states across sequences of operations, as well as the correct order of these sequences, and thereby to validate whether an API is correctly used by the developer.

There are different approaches one can take to create property monitors for OpenSSL certificate validation. For example, we could rely on the OpenSSL documentation to derive the assertions needed. Or, we could study how existing programs use the API and base our property monitor on these examples. Because of the complexity of the APIs and it's documentation, we chose a hybrid approach: based on issues highlighted in previous work and tutorials for using OpenSSL we first build a stylized working example of correct API usage that we then manually validated using the SSL testing site `badssl.com`. We can then use this example to construct property monitors and explore the feasibility of Bounded Model Checking to validate correct API usage. If feasible, we can then apply our technique to real-world examples.

To verify the correctness of application code, we need to model the internal behavior of system and library calls. As we are interested in verifying the correct semantics of OpenSSL API calls, we need to abstract the behavior of these calls. A common approach to do this is to add code calling property monitors to existing program code. For example, for each OpenSSL API function we could insert another function in front of it that takes the same arguments and then updates our property monitor.

We decided to take a slightly different approach. Since all calls we are interested in stem from an external library, we implement header stubs that simulate the behavior of the library, and which CBMC will include instead of the original OpenSSL files. This way, we do not need to modify the original program code to add our property monitors. Hence, we can verify program code by running CBMC using our own OpenSSL header files:

```
cbmc file.c --function main --no-unwinding-assertions --unwind 1 -I ./include
```

A potential downside of this approach is that our property monitors may not capture all of the internal state of OpenSSL. However, due to the limited complexity of certificate validation we are confident that this is not an issue in our case.

We now present a simple property monitor that shows how one can verify the version of the SSL/TLS protocol used when establishing a secure connection. While attacks on outdated protocol versions of SSL mostly target browser software [7], it is nevertheless a good idea to enforce a recent protocol version. In Listing 1, we exemplarily verify that

the protocol version is TLS 1.1. Note the `assert()` function that tells CBMC to validate this condition.

Listing 1: Validation of the TLS Protocol Version

```
inline SSL_CTX *SSL_CTX_new(const SSL_METHOD* method) {
  assert(method == TLSv1_1_client_method());
  SSL_CTX* ctx = (SSL_CTX*)malloc(sizeof(SSL_CTX));
  ctx->method_ = method;
  return ctx;
}
```

4 OpenSSL Property Monitors

We will now first briefly outline the OpenSSL calls necessary to verify that a secure communication channel has been established based on our stylized example (cf. Listing 2). Then, we will explain how our header stubs allow CBMC to validate the code.

First, in lines 9–12, we initialize OpenSSL's internal state. Then, we allocate a SSL context (line 14) that can be shared by multiple connections, an I/O stream abstraction (BIO, line 16–17), and an SSL object, respectively. We enable automatic verification of the hostname in lines 21–22 (corresponding to the procedure available in OpenSSL 1.0.2 [5]) and perform the SSL handshake (line 25).

Next, we check whether the chain-of-trust verification succeeded by comparing the return value of `SSL_get_verify_result` to `X509_V_OK`. However, as `SSL_get_verify_result` will also return `X509_V_OK` when the server did not send a certificate at all, we first need to check that a certificate was indeed presented (lines 31–32).

Listing 2: Stylized Example for Certificate Validation with OpenSSL

```
/* Local variables */
BIO *bio;
SSL_CTX *ctx;
SSL *ssl;
X509 *cert;
X509_VERIFY_PARAM *param = NULL;

/* Initializing OpenSSL */
SSL_load_error_strings();
ERR_load_BIO_strings();
OpenSSL_add_all_algorithms();
SSL_library_init();

ctx = SSL_CTX_new(TLSv1_client_method());

bio = BIO_new_ssl_connect(ctx);
BIO_get_ssl(bio, &ssl);
param = SSL_get0_param(ssl);
```

```
/* Enable automatic hostname checks */
X509_VERIFY_PARAM_set_hostflags(param, X509_CHECK_FLAG_NO_PARTIAL_WILDCARDS
X509_VERIFY_PARAM_set1_host(param, hostname, 0);

/* Verify the connection opened and perform the handshake */
if(SSL_connect(ssl) != 1){
  /* Error during handshake */
  ...
}

/* Check the certificate */
cert = SSL_get_peer_certificate(ssl);
if(cert != NULL){
  if(SSL_get_verify_result(ssl) != X509_V_OK){
    /* Verification failed */
    ...
  }
} else {
  // No certificate
  ...
}

/* Send the request */
BIO_write(bio, request, strlen(request));

/* Read in the response */
int p = BIO_read(bio, r, sizeof(r) - 1);
```

We now describe the verification procedure conducted by our property monitors. The full OpenSSL stub to use with CBMC can be found in our project deliverables, here we will use a graphical representation for better legibility.

In Figure 2 we see the control flow graph that is verified by CBMC. When selecting a protocol version, CBMC checks whether the `SSL_Method` belongs to a set of `Safe_methods`.

When configuring the context or the SSL object, the developer has the option to enable automatic verification of certificates during the handshake using the flag `SSL_VERIFY_PEER`. If automatic verification is set, the developer can provide a callback function that receives `1` if the certificate has passed the chain-of-trust verification, or `0` otherwise. This callback function can be used to return a value of `0` (reject) or `1` (accept) and thereby override the internal validation. Because the callback function can potentially allow to initiate insecure connections (if it always accepts certificates that do not pass the internal test), we explicitly verify whether it accepts invalid certificates.

In sequence, we verify that a data transmission can only occur iff automatic verification was set during the configuration state or if the certificate is not `NULL` after the handshake and `SSL_get_verify_result` returns `X509_V_OK`, which is tracked internally. To validate the hostname, we ensure that the hostname has been set.

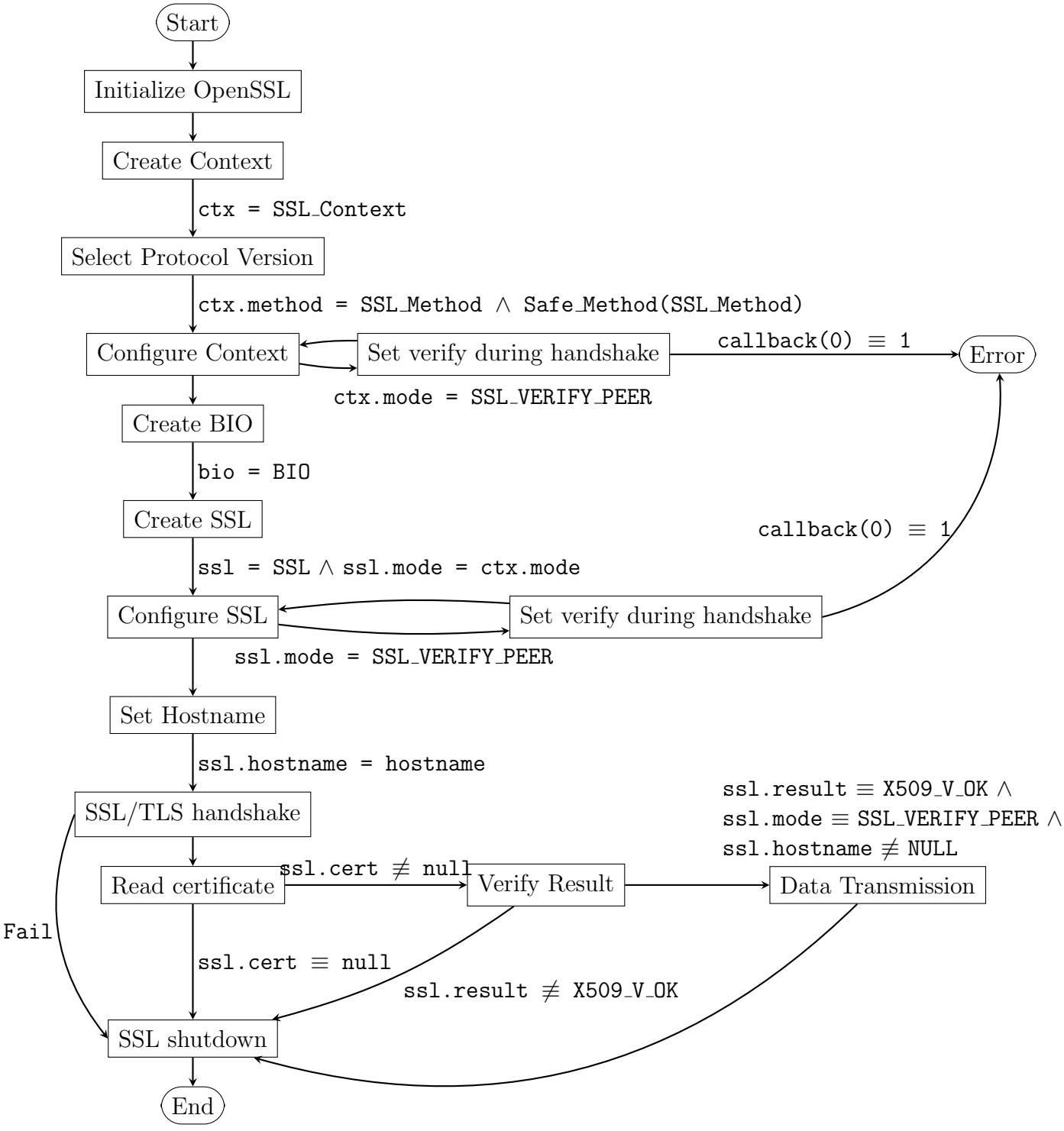

Fig. 2: OpenSSL control flow graph to establish a secure connection.

5 Applying the Property Monitors

The next step is to apply our property monitors to other software projects. When we originally wrote our property monitors in C++, we adapted two simple examples[1] from GitHub as a first starting point. These helped us to ensure that our property monitors are applicable, and both did not include the check for missing certificates. After that, due to some issues with C++ libraries, we modified our property monitors to use C only.

We then tried to apply our property monitors to some old versions of the projects that according to [3] contained API usage bugs in the past. Overall, we had a number of issues (discussed below) with validating these programs using our property monitors and CBMC. These mostly prevented us from employing a structured approach for testing our property monitors on the projects listed in [3].

Use of Other OpenSSL Functionality Our property monitors do not cover all functions that programs may use, e.g., they often import other cryptographic functionality such as hash functions. Our solution usually was to remove any other OpenSSL include statements.

Dependencies and Symbol Definitions Projects often had dependencies that are irrelevant for certificate validation, but whose (missing) header files CBMC would try to import. When we removed these include statements, we sometimes ended up with missing symbol definitions, where we either had to manually add these definitions or remove the code sections that used these symbols.

Optional SSL Usage Some codebases make the use of SSL optional using conditional groups. For example, the spamc library of SpamAssassin requires us to pass the option `-D SPAMC_SSL=true` in order for CBMC to validate the SSL code.

Aborting In some programs we found usage of functions outside of the main files that we tested, which would log debug information and then abort the program. If we did not pass these additional files to CBMC, it will assume that the program does not terminate and thereby may incorrectly report validations.

CBMC Library Abstractions CBMC provides internal abstraction for the string functions in `string.h`. Strangely, CBMC would not validate beyond these functions, but report all following assertions as `SUCCESS`. We were not able to debug this issue and added replacement functions that return nondeterministic values as a temporary fix to allow validation. (Another possibility should be to disable all internal library abstractions with the option `--no-library`, but this did not produce correct results for our stylized example).

Project Size We performed our experiments in a virtual machine equipped with 4GB of memory. This was not sufficient for some of the projects we tried (e.g., picolisp).

[1] `https://github.com/yunuscanemre/cpsc526/blob/master/hw1/c/sslFtp/client/ftpClient.cpp`
`https://github.com/LaurieHarding-Russell/Prometheus/blob/master/prometheus/internet.cpp`

Tab. 1: Results for the verification of different software projects

Project	Version	Results	Issues/Notes
spamc	3.3.2	uses old protocol version missing verify result check missing hostname check	requires `-D SPAMC_SSL=true`
ratproxy	1.58	uses old protocol version missing verify result check missing hostname check	requires `--nondet-static`
dma	0.9	—	compilation fails
picolisp	3.1.5.2	—	runs out of memory

Let us highlight these issues with the example of the `spamc` module of the SpamAssassin spam filter, for which we retrieved an old version (3.3.2) from Ubuntu launchpad[2]. First, we need to run the configure script to generate relevant config files. Next, we remove two OpenSSL imports from `util.h` that produced syntax errors in CBMC. As various string functions are used in this file, our temporary replacement of these functions are also needed for CBMC to correctly validate the code. Now, we can finally run CBMC on the file `libspamc.c`. The failed verification results CBMC reports missing validation of the chain-of-trust verification as well as missing hostname validation.

We report the results of the examples that we have included in our project deliverables in Table 1. These were taken from [3], however missing information about the version of the software made exact reproducibility difficult.

6 Conclusion

In our project we evaluated the feasibility of verifying correct API usage of the cryptographic library OpenSSL using Bounded Model Checking in combination with property monitors. On the one hand, we were able to show the feasibility of our approach based on a stylized example that we built based on relevant related work. Our property monitors are able to detect the use of obsolete protocol versions, mistakes made when verifying the chain-of-trust as well as missing hostname validation. On the other hand, we encountered various limitations of our approach when we tried to apply it to other software, which makes it currently difficult (and sometimes frustrating) to use.

While it might be feasible to improve our approach and automate some of the fixes we discussed in the last section (especially for someone with more familiarity of the C programming language), this bears the question whether the effort is beneficial. Ultimately, the scenario at hand is rather simple and previous work [3, 8, 2] has pointed out the most common errors developers make in their implementations. Furthermore, the amount of internal state that our property monitors must keep track of is limited. In many cases a manual inspection of the source code was easier than applying our property monitors. In this case, more abstract solutions (e.g., [3, 8]) may provide a better approach to verifying these issues since they are able to omit some of our technical problems. A complementary

[2] `https://launchpad.net/ubuntu/+source/spamassassin`

solution would be to have better abstractions of the cryptographic procedures themselves (i.e. as a software package that people use instead of OpenSSL). To construct this, our property monitors could be helpful to prove correctness.

An open question remains whether other areas of API usage might benefit more from our approach. Initially, we evaluated a few related ideas such as the verification of digital signatures, but discovered that most of these validations are not exposed by the API of cryptographic libraries. We also evaluated the feasibility of synthesis to create our property monitors, but the limited state we need to keep track of reduces the utility of such an approach.

To conclude, this project revealed the challenges in writing secure code using cryptographic libraries, and also the challenge in countering these with model checking techniques. Nevertheless, more research in this direction is certainly helpful and we encourage others to follow our example.

References

[1] Edmund Clarke, Daniel Kroening, and Flavio Lerda. "A Tool for Checking ANSI-C Programs". In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*. Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, 2004, pp. 168–176. ISBN: 3-540-21299-X.

[2] Martin Georgiev et al. "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*. ACM. 2012, pp. 38–49.

[3] Boyuan He et al. "Vetting SSL usage in applications with SSLint". In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 519–534.

[4] Yabing Liu et al. "An End-to-End Measurement of Certificate Revocation in the Web's PKI". In: *Proceedings of the 2015 ACM Internet Measurement Conference*. ACM. 2015, pp. 183–196.

[5] OpenSSL Wiki. *Hostname Validation*. 2016. URL: `https://wiki.openssl.org/index.php?title=Hostname_validation&oldid=2359` (visited on 11/22/2016).

[6] Wikipedia. *OpenSSL — Wikipedia, The Free Encyclopedia*. 2017. URL: `https://en.wikipedia.org/w/index.php?title=OpenSSL&oldid=758980829` (visited on 01/08/2017).

[7] Wikipedia. *Transport Layer Security — Wikipedia, The Free Encyclopedia*. 2017. URL: `https://en.wikipedia.org/w/index.php?title=Transport_Layer_Security&oldid=759082389#Attacks_against_TLS.2FSSL` (visited on 01/09/2017).

[8] Insu Yun et al. "APISan: Sanitizing API Usages through Semantic Cross-checking". In: *Proceedings of the 25th USENIX Security Symposium (Security)*. Austin, TX, 2016.