# CRYScanner: Finding cryptographic libraries misuse

Amit Choudhari
*Institut Polytechnique de Paris*
Palaiseau, France
amit.choudhari@polytechnique.fr

Sylvain Guilley
*Secure-IC S.A.S*
Paris, France
sylvain.guilley@secure-ic.com

Khaled Karray
*Secure-IC S.A.S*
Paris, France
khaled.karray@secure-ic.com

***Abstract*—Cryptographic libraries have become an integral part of every digital device. Studies have shown that these systems are not only vulnerable due to bugs in cryptographic libraries, but also due to misuse of these libraries. In this paper, we focus on vulnerabilities introduced by the application developer. We performed a survey on the potential misusage of well-known libraries such as PKCS #11. We introduced a generic tool CRYScanner, to identify such misuses during and post-development. It works on the similar philosophy of an intrusion detection system for an internal network. This tool provides verification functions needed to check the safety of the code, such as detecting incorrect call flow and input parameters.**

**We performed a feature-wise comparison with the existing state of the art solutions. CRYScanner includes additional features, preserving the capabilities of both static and dynamic analysis tools. We also show the detection of potential vulnerabilities in the several sample codes found online.**

***Index Terms*—Cryptography libraries, misuse, dynamic analysis, novel "CRYScanner" tool, CWE-1240.**

## I. Introduction

In today's digital world, security and privacy are growing concerns for every user. To satiate this need, many user applications use cryptographic algorithms for authentication, storing data and communication. Open-source libraries such as OpenSSL, OpenSSH, SoftHSM (OASIS PKCS #11) provide a means to use these algorithms. Even though in theory the cryptographic algorithm guarantees confidentiality, integrity and authentication, the implementation of these algorithms could have bugs. For instance, the heartbleed bug (CVE-2014-0160) found in OpenSSL's implementation of TLS leads to a leakage of memory contents from the server to the client [3]. Such issues in the implementation of standards could jeopardize the whole system. On the other hand, these libraries allow several, arguably a lot, modes and options which in turn leaves room for the developer to make mistakes. For instance, PKCS #11 provides options such as CKM_XOR_BASE_AND_DATA [1] that are vulnerable to related-key attacks. In this paper, we focus on detecting such vulnerabilities that could be caused due to a misuse introduced by the developer (as described in MITRE CWE-1240: "*Use of a Cryptographic Primitive with a Risky Implementation*").

Lazar et al. [10] studied 269 cryptographic vulnerabilities reported in the CVE database. They found that 88% of these were due to misuse of cryptographic libraries by individual applications. In another research by Patnaik et al. [16], an extensive analysis of 2400 questions on 7 cryptographic libraries was conducted. It was found that the majority of questions were around the categories such as missing information, not knowing what to do, not knowing if it can be done, and issues across time and space. The underlying reason is insufficient documentation, lack of examples, API level abstraction. It suggests that the developer is not well equipped with the necessary information and struggles with the usage of most cryptographic libraries. To fill this gap, it seems quite intuitive to have a tool that assists the developer and maintainer, during the development and post-development stages.

Researchers have come up with various tools for static analysis [9] [8] [19], compile-time analysis [22] and dynamic analysis [17] to detect cryptographic misuse. Static analysis tools can check the sanity of the program with different possible combinations of arguments. It allows the tool to verify the potential paths a program could take during run-time. The drawback of static analysis is that it has many false positives [17]. Also, it is prone to miss execution paths that are more likely to be taken during runtime. The compile-time analysis tools like Anselm [22] are programming language agnostic. It checks for acceptable and unacceptable call flows using LLVM intermediate representation. Dynamic analysis tools such as CRYLOGGER [17], SMV-hunter [20], iCryptotracer [11] are platform dependent. It allows testing a system with real-world inputs in a particular scenario. But, they are tightly coupled to a programming language and their respective platform.

In this paper, we present "CRYScanner", an open-source tool. It runs in two stages, the runtime stage to collect logs and the offline stage to analyze these logs. The tool in general is library agnostic. CRYLOGGER verifies the input, output of cryptographic lib API also verifies the expected call flow. The tool inherits properties of both static and dynamic analysis tools. It is further discussed in detail in section IV.

## II. Cryptographic library misuse

Cryptographic libraries are in a process of constant evolution for supporting newer cryptographic algorithms, introducing better abstractions, fixing security bugs. The international standard ISO/IEC 29128:2011 standard tracks requirements in this respect. These newer versions of libraries released frequently gives rise to (1) Backward compatibility, to ensure no breakage of existing applications; (2) Insufficient documentation with examples; (3) Same library handling multiple

Keynote talk at IEEE NICS 2021.

operations, giving too much control to a developer. In this section, we shall study misuses in different libraries suffering from these issues.

### A. PKCS #11

Public Key Cryptography Standards (PKCS) were developed by 'RSA.inc' to provide compatibility and interoperability between devices and implementations. PKCS #11 covers Cryptographic Token Interface standard for accessing cryptographic stores such as hardware security module (HSM). These cryptographic stores also called a token, stores objects such as keys, data, certificates and performs cryptographic operations. Each object has a set of properties that determine its behaviour. For example, sensitive keys can never get extracted outside the token in cleartext.

Research by Clulow studied that the key-management APIs, when incorrectly used, are prone to attacks causing leakage of keys [6]. In asymmetric encryption, when an encryption key is generated it is never exported outside the device in cleartext. A Key Encryption Key (KEK) is used to wrap the key. Bond in [5] has pointed to an attack of key conjuring, where a new untrusted key is injected into the device. For instance, a random bit sequence passed as an input to `C_Unwrap()`, adding a new untrusted, weak key to the token. Using weaker algorithms and keys are a common issue when the security module of an application is not updated periodically. Attacks such as Key binding and Key separation attacks are mostly due to insufficiently documented examples and poor use of API. Few attributes of the key objects such as extractable, operation type, should be immutable. For instance, modifying a property from unwrapping to decrypt, while creating the copy of object `C_CopyObject()`, will allow leakage of the key in cleartext. Attacks such as reduced keyspace and related keys are possible due to vulnerable modes and options. For instance, the option CKM_XOR_BASE_AND_DATA allows users to derive new keys by XORing with known bytes. The adversary generates several such related keys and then performs an exhaustive search on it with known text. If even one key is found, the original key can be computed by XORing with the related bytes. In PKCS #11 asymmetric encryption, a public key is exported in clear and hence is prone to additional attacks like injecting Trojan public and wrapped key [6].

### B. Padding attacks

A padding attack is a side-channel attack where the attacker has access to an oracle that returns true for a chosen-ciphertext when the padding is correct. For instance, Vaudney's attack on Cipher Block Chaining (CBC) mode symmetric key encryption using RC5 padding scheme [21]. The attacker retries with a chosen padded section to correctly identify the length of the ciphertext. Bleichenbacher's attack is a similar kind of padding attack [4]. PKCS is found to be vulnerable to such attacks, and hence it is necessary to use safer padding techniques like PKCS #1 OEAP padding.

## III. State of the art Methodology to detect misuse

As studied by Paterson et al [15], a gap exists between the way cryptography is studied in theory, the way standards are defined, developers implement the software, and users consume it. For this reason, it is recommended to use the cryptographic algorithms that are approved by a standard authority such as the National Institute of Standards and Technology (NIST). These algorithms undergo extensive security analysis and are periodically reassessed for continued effectiveness.

### A. Cryptoanalysis

An attack from a Theoretical cryptographer's perspective is very different from a developer. Theoreticians are concerned about the indistinguishability of the ciphertext, true randomness of the nonce, attacks requiring a huge number of cleartexts. These attacks are not cared much by a developer as they seem to exist only on paper and there is no real demonstration possible. For instance, there are several potential theoretical attacks reported for AES [7]. However, due to a lack of pragmatism, it is still considered safe by NIST. As the majority of security issues are found in the implementation of cryptographic algorithms the focus has shifted from theoretical cryptography to implementation of it.

### B. Crypto implementation

In this section, we will evaluate the state of art solutions that try to address the misuses II from both static and dynamic analysis perspectives.

*a) CRYLOGGER:* Android applications use Java cryptographic algorithms (JCA) to perform cryptographic operations like authentication, storing the data, checking integrity. CRYLOGGER [17] is designed to detect API misuses of JCA through dynamic analysis. The functionality is split into two phases Online logger and Offline checker [Figure: 1]. The Online logger monitors the API calls, the arguments passed and logs this information into a file. The offline checker scans the logs file to identify any known violation of crypto rules. It classifies the cryptographic libraries into 7 classes Message digest, symmetric encryption. asymmetric encryption, key derivation/generation, random number generation, key storage, SSL/TLS and certificates.

The major drawback of CRYLOGGER is the difficulty to set up and the lack of flexibility of the tool. The online logger needs a wrapper library over the JCA to print logs. Any update in the JCA library requires an update to the wrapper as well. The addition of a new cryptographic library with similar needs cannot be tested without creating a new wrapper for it. This adds another overhead of maintenance and versioning. Unlike static analysis tools, it is not capable of detecting incorrect usage of API orders.

*b) CRYSL and CogniCrypt$_{SAST}$:* Developers are not always well equipped with sufficient examples to use cryptographic operations and end up developing weak software. It is best if the bug is caught at the development stage. CRYSL [9] is a definition language that enables cryptography

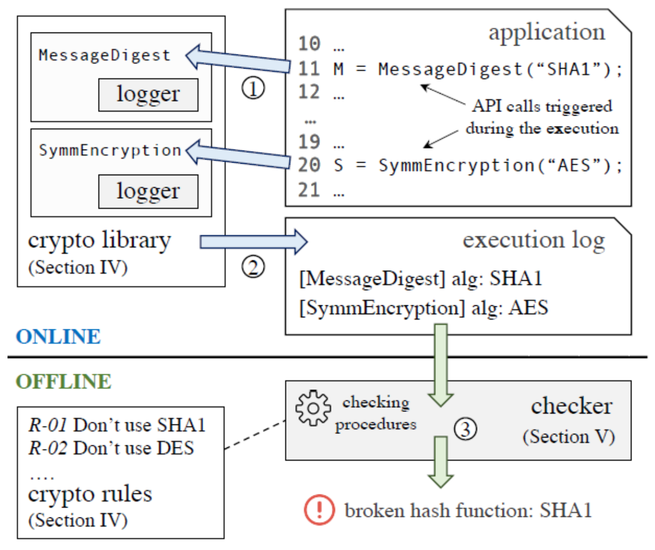

Fig. 1. CRYLOGGER block diagram (courtesy of [17])

experts to define the correct use of their cryptographic library. CogniCrypt$_{SAST}$ [8] is a static analysis tool, that uses CRYSL rules to guide the developer during the development stage of the application. It is available as a plugin in eclipse.

Being a static analysis tool, CRYSL inherits the limitation to analyse runtime values and behaviour of a system. Weak passwords set during runtime could compromise the systems completely. The tool fails to find misuse if there are multiple levels of abstraction before using JCA API. It fails to detect replay attacks due to hardcoded or reused values. Even though the idea is generic, the tool implementation in itself is very tightly coupled to the Android Java platform. Applying this on other platforms like Linux needs a complete rework.

## IV. Contribution

As discussed in section II, if cryptographic APIs are not used diligently, it could compromise the security entirely. Moreover, it is challenging to find the updated and correct usage of the APIs. Developers eventually end up referring to unofficial sources such as StackOverflow and other repositories. Which in turn result in making similar mistakes across several applications. Hence, instead of relying on the examples and documentation completely, we propose a tool that will allow the cryptographic experts to define strict rules for their given library. The developers could use this to verify the runtime behaviour of their applications.

### A. CRYScanner Syntax

CRYScanner is designed to read heterogeneous logging information and parse rules with well-defined syntax. In this section, we will discuss design decisions and their influence on the syntax for creating rules.

*1) Design decisions:*

*Runtime logging:* While comparing the state of the art tools we clearly saw a gap in capabilities between static, dynamic and compile-time analysis III. To devise a tool that is generic enough as CRYSL and also could analyse runtime behaviour, we decided to use the GNU debugger (GDB) debugger to log the information. This makes our tool generic across different applications and libraries.

*Whitelisting:* It is well known that there are many ways for a system to fail, in fact, there are always newer vulnerable configurations discovered. But, only a few sequences are safe for a user to use. Hence, we decided to use whitelisting approach to define the rules for a cryptographic library.

*Storage type independent:* All the rules are expected to be written by a cryptographic expert who is aware of the expected values. In that case, there is no requirement to add a concept of storage type in the rules file. The parser only has to compare the values between the log file and the rule file.

*API call order and constraints:* While studying the potential misuse across different libraries in section II, we concluded that the majority of bugs are observed due to APIs invoked in improper order and incorrect parameters passed to the API. For instance, the context object not freed in a conditional case will leak information. To address these concerns, we focused our tool on API call order and values passed to it.

*Generic rule parser:* In any software lifecycle, maintenance and upgrade is a sizable ongoing task. Given the fact that cryptographic libraries undergo periodic version updates, on top of that, a tool upgrade would be overkill. Hence, we aimed for a generic tool that requires minimal change across version upgrades and adding support for the newer library.

*2) Rule sections:* Each cryptographic library has a rule file defined for it. The rules are a set of platform-independent instructions, split in sections, defining the expected behaviour of a program.

Listing 1
CRYScanner sample rule file for AES encryption

```
OBJECTS
        a: EVP_CIPHER_CTX_new()
        b: EVP_CIPHER_CTX_free(c)
        c: EVP_EncryptInit_ex()
        d: EVP_EncryptUpdate()
        e: EVP_EncryptFinal_ex()

ORDER
        (ab)*
        (a(cd+e)b)*

CONSTRAINTS
        EQ(e:ctx.key_len==32)

FORBIDDEN
        c
```

`OBJECTS`: The chosen mode of debugging is a dynamic analysis and we are aware that the function calls are the only entry point for an application to request service to a library. Hence, APIs defined in the `OBJECTS` section are the fundamental blocks around which the rules

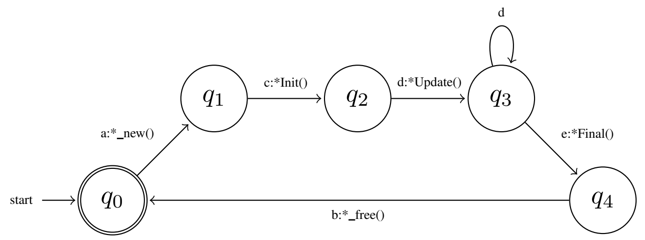

Fig. 2. State machine for the Encryption order defined in Listing 1

are defined. Syntax of an object is {pseudo_name}: {function_name}({parameter_name,...}). These objects are referenced using the pseudo_names from other sections. The rule parser creates an object with function_name as the key for every function defined. As shown in Listing 1, each OBJECT declaration contains signature of a function name and its variables, of form $(m, v)$, where m $\in M$ and v $\in V^*$. This object internally contains a map of variables and their respective value.

*ORDER:* Regular expression of pseudo names is used to define the usage of a particular operation. The syntax for the order is an aggregation of OBJECTS from $M$. Pseudo names are used since it is easier to read and write a regular expression. For instance, in Listing 1, the EVP_* APIs are always expected to follow the pattern of *Init(), followed by multiple *Update() and end with *Final() call. To ensure this order is followed, the regular expression is defined as (a(cd+e)b)⋆. The parser internally converts this regular expression into an equivalent deterministic finite state machine [Figure: 2]. Let, $Q$ be a set of all states, $q0$ be the initial state, and $F$ be a set of final states. The automaton can thus be defined as $(Q, M, \delta, q0, F)$, where the state transition matrix is $\delta : MXQ$. A violation is detected if the state machine has not reached its final state by the end of the program.

*CONSTRAINTS:* As seen in section II, API's can be misused by sending undesired values through arguments. CONSTRAINTS section is used to keep a check on such misuse. Each Constraint is a Boolean function where the argument is a function that maps the variable V to object O, such that $C := (v \rightarrow O \cup V^*)$. The syntax for defining a constraint is {operation}({operand}{equation}⋆optional). where operation is equality check 'EQ()', primality check 'PRIME()', replay check 'REPLAY()', randomness tests 'RAND()'. The equality check supports $>, <, <=, >=, ==$ operators, with rvalue being a number, string or set of string. As we are following the whitelisting approach, only allowed values need to be mentioned. For instance, if the key size should be larger than 32 for encryption function, then the constraint should be defined as EQ(c:key > 32). Generating prime numbers is a common and important step in cryptography. Primality check allows the developer to verify whether the number passed was prime or not. Replay check can be used to avoid replay attack or any kind of reuse of keys, salts, an initialization vector (IV) and nonce. Reusing the same salt for multiple hashing operations is known to be vulnerable to rainbow table attack [2].

TABLE I
USAGE OF SUPPORTED OPERATIONS IN CONSTRAINTS SECTION

| Operation | Description |
|---|---|
| EQ(b:pad == RSA_PKCS1_PADDING) | Compares the string 'RSA_PKCS1_PADDING' argument pad. |
| PRIME(b:rsa_p) | Verifies the primality of argument rsa_p in function b |
| REPLAY(b:key) | Checks for any reuse of the variable key across multiple runs. |
| RAND(b:salt) | Runs NIST tests for random number generators on argument salt of API b. |

*FORBIDDEN:* Deprecating APIs due to known vulnerabilities is a common practice over version upgrades. The library continues to allow the usage of such API to support backward compatibility. This does not force the user nor does it notify the maintainer causing non-functional regression. FORBIDDEN class is the only section to follow the blacklisting approach. The syntax for declaring forbidden APIs is to list the corresponding comma separated pseudo names.

A rule is a tuple $(M, F, A, C)$, where $M$ is list of function declarations in OBJECTS, $F$ is the list of forbidden functions, $A$ is the deterministic finite state automaton $(Q, M, \delta, q0, F)$, and $C$ is the list of constraints.

## B. Runtime logger

Dynamic analysis tools need enough run-time trace to perform analysis. We are looking for a generic solution for tracing logs and hence we decided to use a debugger with a python script instead of creating a wrapper library with additional debugs. GDB one of the most common tool for debugging has all the capabilities needed for run-time tracings, such as adding breakpoints on API calls, printing stack trace, printing arguments of different storage types passed to the API. We developed an easy to use python script, that will break at all the entry points listed by the expert and keep printing the necessary information till the end of the program. The generated log file is then given as an input to the offline rule checker to identify violations.

## C. Implementation

CRYScanner tool execution is split into two stages, (1) Online logger and (2) Offline rule parser [Figure: 3]. The rule and log parser uses an open-source parsing library pyparser [12]. In the first stage, the tool collects the run-time logs of the program and pre-process them into a structured format. In the second stage, these pre-processed logs are parsed into a list of OBJECTS types. The rule file is parsed to create goal-oriented objects like CONSTRAINTS, ORDER, FORBIDDEN for verification.

The tool is designed such that class Object is the basic building block. Individual types of rule operations are split as per their classes. The class ORDER uses opensource

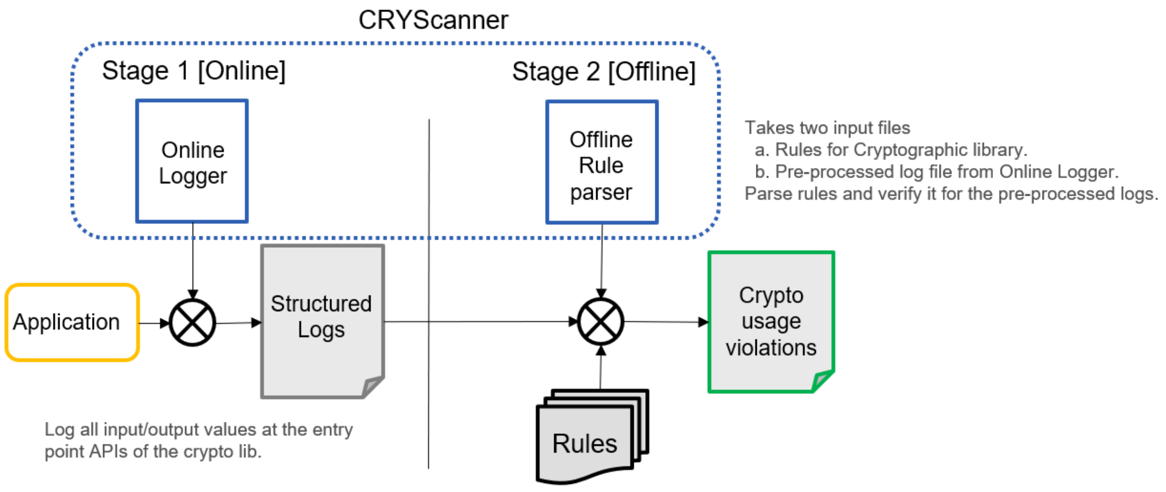
CRYScanner
Stage 1 [Online]
Stage 2 [Offline]
Online Logger
Offline Rule parser
Takes two input files
a. Rules for Cryptographic library.
b. Pre-processed log file from Online Logger.
Parse rules and verify it for the pre-processed logs.
Application
Structured Logs
Crypto usage violations
Rules
Log all input/output values at the entry point APIs of the crypto lib.

Fig. 3. CRYScanner execution flow diagram

libraries such as greenery for processing the regular expression [18] and finite-state machine-building lib such as transition [13]. For testing random numbers we used NIST Test Suite for RNG from library [14]. overall the tool is designed in a modular style, so it is very easy to extend the capability of the Rule file. Supporting more constraint checks can be added easily with the existing tokens framework. For example, adding different randomness tests. The source code for the tool along with sample rule files, logs and README file is available at https://github.com/amitsirius/cryscanner.

### D. Evaluation

We evaluate the CRYScanner tool based on its capability to detect different types of misuses, precision to detect all known vulnerabilities, performance based on verification time and the comparison with state of the art.

#### 1) Types of misuses:

*Setup:* As discussed in section IV-A2, the violations are detected using functions such as equality (`EQ`), replay (`REPLAY`), randomness (`RAND`), prime (`PRIME`), order (`ORDER`) and forbidden objects (`FORBIDDEN`). We randomly identified code snippets from StackOverflow and public git repositories as a sample set. These snippets were mostly marked as accepted answers or upvoted. For ease of writing rules, we picked the code for similar operations like symmetric and asymmetric encryption techniques.

*Result:* After running the tool over 10 different code snippets, the tool could identify 35 violations. As seen in Figure 4, the most common misuse is replay attack, i.e reuse of same keys, salts, IV. As most of the examples were related to encryption techniques, it is understandable to find more hits on replay. The randomness function tests the bits sequence on 10 different NIST tests such as mono bit, longest run, discrete Fourier transform. We decided on a threshold of at least 6 tests to pass. Order misuse was observed due to a missing function for clearing context, which could lead to leakage of information. The equality checks detected violations such as

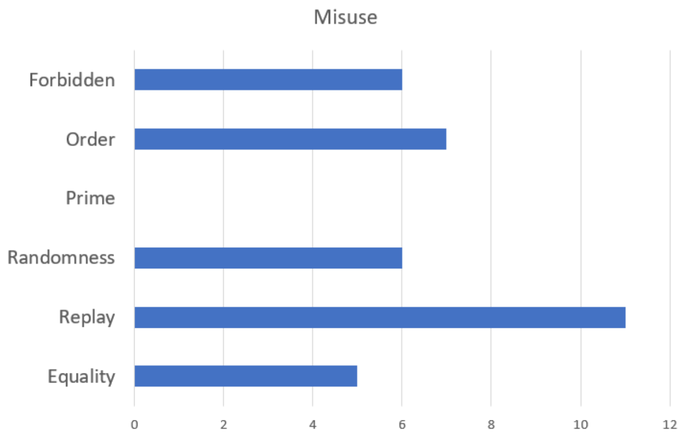

Fig. 4. CRYScanner tool class diagram

smaller keys, weaker padding schemes. The forbidden checks identified usage of deprecated functions and non recommended low-level API. The primality check was functionally tested, but the samples did not contain any potential use-case.

#### 2) Performance:

*Setup:* The total execution time for the tool is split into the log parsing and verification stage. The log parsing stage depends on several relevant cryptographic operations performed. The verification time required for every application is related to factors such as relevant objects and their corresponding function in rules. Each function takes a different time. For example, primality and equality check is performed on every instance of the object. Whereas randomness and order tests are run on all instances combined. While testing the performance we ran a program with different encryption operations for 20 iterations. It allows us to get more quantifiable numbers on each section.

*Result:* Table II shows that the majority of execution time is spent on parsing the logs and run-time logger. All other verification requires negligible duration. Even with all

TABLE II
PROFILING SUMMARY OF EACH BLOCK WHEN AN ENCRYPTION DECRYPTION APPLICATION WAS RUN 20 TIMES.

| **Function** | **Execution Time (sec)** |
|---|---|
| **Run-time logger** | 174 |
| **Parsing Logs** | 560 |
| **Parsing rules** | 0.23 |
| **Constraints check** | 11.13 |
| **Order check** | 11.52 |
| **Forbidden check** | 11.2 |
| **Total** | 768.08 |

TABLE III
FEATURE COMPARISON BETWEEN STATE OF THE ART AND CRYSCANNER.

| **Capability** | **CogniCrypt** | **CRYLOGGER** | **Anselm** | **CRYScanner** |
|---|---|---|---|---|
| Call flow | ✓ | | ✓ | ✓ |
| Code static analysis | ✓ | | ✓ | |
| Run-time values | | ✓ | | ✓ |
| Detect replay | | ✓ | | ✓ |
| Immune to abstraction | | | | ✓ |
| Immune to library update | ✓ | | | ✓ |
| Library agnostic | ✓ | | ✓ | ✓ |
| Platform Independent | | | ✓ | ✓ |
| Remote analysis | | | | ✓ |

this, a single run of an application using the tool takes less than 2 minutes on average. There will be a significant performance improvement (approximately 20%) if the log parsing is performed in parallel with the run-time logger.

*3) Precision:* The precision of a tool is determined based on factors like a true positive, true negative, false positive and false negative. As the tool is still in the preliminary stage, black-box testing on applications is not feasible. Hence we evaluated the possibilities of the tool hitting false positives and false negatives. Sometimes the API has a default behaviour when the user passes zero or null as an argument. Such behaviour is very API specific and at present, the tool is not capable of identifying such complicated behaviour leading to false positives and false negatives.

*4) Comparison to state of art:* From Table III, it can be seen that the tool is designed to inherit the capabilities from both dynamic and static analysis tools. Call flow verification is used to detect unexpected divergent call sequences of an application. This capability is mostly observed in static analysis tools, but CRYScanner also performs the same verification building a finite state automaton. Similarly, checking the real-time values is a quite powerful function for identifying any violation from inputs and common execution paths. Unlike other dynamic tools, CRYScanner is aimed to be generic across different platforms (Linux, Windows) and libraries (cryptoki, OpenSSL). As it uses a debugger to add breakpoints, it is not possible to miss any API call and hence it is immune to any type of abstraction in the code.

## V. FUTURE SCOPE

A dynamic analysis tool has inherent challenges such as the difficulty of setup, limited view of the system, inability to trace uncommon code paths. These issues cause an increase in false negatives. We have identified room for improvement of the tool as per the categories **Ease of usability**: For a developer, an analysis tool must need minimum configuration. With the help of cryptographic experts, create an elaborate rule set for well-known libraries such as OpenSSL. It reduces the overhead for developers and the possibility of making mistakes. Adding gdbserver support will allow remote debugging. **Precision and Robustness**: The testing framework should trace all the common paths and be equally capable to execute unusual scenarios. To justify the precision and robustness of the tool it needs to undergo testing with fuzzing and monkey testing framework. **Performance**: The majority of time is consumed while parsing and collecting logs. Running both stages in parallel will improve the performance. **Feature enhancement and hard problems**: Every new feature enhances the reach of identifying complicated bugs. Adding support for context information, function overloading, an array of structs, double pointers will provide richer control in the tool.

## VI. CONCLUSION

We presented a generic tool to detect potential vulnerabilities while using cryptographic libraries. We have successfully tested the tool across different cryptographic operations and libraries [Section IV-D]. In reality, application developers are not always cryptographic experts. This tool is designed for developers to require limited to no knowledge on the details of the complex algorithm. We have empirically evaluated CRYScanner from the application developer and maintainer point of view. The execution time required for the tool to analyse longer runs is less than 2 minutes [Table II]. This makes it easier for a maintainer to run across hundreds of applications with little overhead. Often, it is more tricky to perform analysis on the live system due to a lack of tools. CRYScanner fits perfectly in such cases, the developer could use gdbserver and analyse the running application remotely. When compared to any other state of the art solution, CRYScanner outperforms in feature and flexibility aspects. We want to approach the cryptographic experts of the crypto libraries and collect information for building rules. With this, we encourage application developers to use this tool to secure applications.

## REFERENCES

[1] Safenet protecttoolkit-c mechanisms. SafeNet ProtectToolkit 5.9. URL: https://thalesdocs.com/gphsm/ptk/5.9/docs/Content/PTK-C_Program/PTK-C_Mechs/CKM_XOR_BASE_AND_DATA.htm.

[2] Salted Password Hashing - Doing it Right. crackstation.net. URL: https://crackstation.net/hashing-security.htm.

[3] CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160., 03 2014. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160.

[4] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. efficient padding oracle attacks on cryptographic hardware. 04 2012.

[5] Mike Bond. Attacks on cryptoprocessor transaction sets. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems — CHES 2001*, pages 220–234, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[6] Jolyon Clulow. On the Security of PKCS#11. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2003.

[7] Herman Isa, Iskandar Bahari, Hasibah Sufian, and Muhammad Zaba. AES: Current security and efficiency analysis of its alternatives. pages 267–274, 12 2011.

[8] Stefan Kruger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Gopfert, Felix Gunther, Christian Weinert, Daniel Demmler, and Ram Kamath. CogniCrypt: Supporting developers in using cryptography. pages 931–936, 10 2017.

[9] Stefan Kruger, Johannes Spath, Karim Ali, Eric Bodden, and Mira Mezini. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. *IEEE Transactions on Software Engineering*, PP:1–1, 10 2019.

[10] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail? A case study and open problems. *Proceedings of 5th Asia-Pacific Workshop on Systems, APSYS 2014*, 06 2014.

[11] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. pages 349–362, 10 2014.

[12] Paul McGuire. Python library for creating PEG parsers. https://github.com/pyparsing/pyparsing, 2020.

[13] Alexander Neumann. pytransitions. https://github.com/pytransitions/transitions, 2020.

[14] Luca Pasqualini. Nistrng, 2020.

[15] Kenneth G. Paterson and Arnold Yau. Cryptography in Theory and Practice: The Case of Encryption in IPsec. volume 2005, page 416, 01 2005.

[16] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. Usability Smells: An Analysis of Developers' Struggle with Crypto Libraries. In *Proceedings of the Fifteenth USENIX Conference on Usable Privacy and Security*, SOUPS'19, page 245–257, USA, 2019. USENIX Association.

[17] Luca Piccolboni, Giuseppe Di Guglielmo, Luca Carloni, and Simha Sethumadhavan. CRYLOGGER: Detecting Crypto Misuses Dynamically. 07 2020.

[18] "qntm". FSM/regex conversion library. https://github.com/qntm/greenery, 2020.

[19] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects. pages 2455–2472, 11 2019.

[20] David Sounthiraraj, Justin Sahs, Garrett Greenwood, Zhiqiang Lin, and Latifur Khan. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. 01 2014.

[21] Serge Vaudenay. Security flaws induced by CBC padding - Applications to SSL, IPSEC, WTLS... volume 2332, 04 2002.

[22] UCLA William Wang. anselm. https://github.com/trailofbits/anselm, 2020.