SAMBA: Detecting SSL/TLS API Misuses in IoT Binary Applications

Kaizheng Liu[†], Ming Yang[†], Zhen Ling[†*], Yuan Zhang[‡], Chongqing Lei[†], Lan Luo[§] and Xinwen Fu[¶]
[†]Southeast University, China, Email: {kzliu18, yangming2002, zhenling, leicq}@seu.edu.cn
[‡]Fudan University, China, Email: yuanxzhang@fudan.edu.cn
[§]Anhui University of Technology, China, Email: lluo@ahut.edu.cn
[¶]University of Massachusetts Lowell, Lowell, MA, USA, Email: xinwen_fu@uml.edu

***Abstract*—IoT devices are increasingly adopting Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols. However, the misuse of SSL/TLS libraries still threatens the communication. Existing tools for detecting SSL/TLS API misuses primarily rely on source code analysis while IoT applications are usually released as binaries with no source code. This paper presents SAMBA, a novel tool to automatically detect SSL/TLS API misuses in IoT binaries through static analysis. To overcome the path explosion problem and deal with various SSL/TLS implementations, we introduce a three-level reduction method to construct the SSL/TLS API-centric graph (SAG), which has a much smaller size compared with the conventional inter-procedural control flow graph. We propose a formal expression of API misuse signatures, which is capable of capturing different types of misuse, particularly those in the SSL/TLS connection establishment process. We successfully analyze 115 IoT binaries and find that 94 of them have the vulnerability of insecure certificate verification and 112 support deprecated SSL/TLS protocols. SAMBA is the first IoT binary analysis system for detecting SSL/TLS API misuses.**

***Index Terms*—Internet of things, SSL/TLS, Binary analysis**

I. INTRODUCTION

IoT devices are increasingly adopting secure network communication protocols such as the Secure Sockets Layer (SSL) protocol and the Transport Layer Security (TLS) protocol, which are complex and prone to misuse. Misuses of SSL/TLS APIs have exposed applications to various severe security risks [1], [2]. There are two common types of vulnerabilities caused by the SSL/TLS API misuse: *incorrect certificate verification* and *support of deprecated protocol*. These vulnerabilities can be exploited by various attacks such as the man-in-the-middle (MITM) attack and POODLE attacks [3], which are downgrade attacks against SSL/TLS.

Great efforts have been devoted to detecting insecure SSL/TLS API use in network applications. Those works either statically analyze the source code of an application [1], [2] or use dynamic analysis to identify insecure SSL/TLS connections [4], [5]. However, these approaches cannot be readily applied to IoT devices. First, it is still a challenge to emulate IoT binaries due to dependencies on specific configurations of hardware (such as cameras and sensors [6], [7]) despite research on emulation [8], [9] and hardware-in-the-loop rehosting [10], [11]. Therefore, dynamic analysis may not be unsuitable for detecting insecure SSL/TLS connection establishments for IoT binaries. Second, IoT applications are usually released as binaries with no source code.

* Corresponding author: Prof. Zhen Ling of Southeast University, China.

Detecting SSL/TLS API misuses in IoT binaries requires novel static analysis techniques. First, there is an efficiency issue. Detecting such misuses often requires static analysis of control flow and data flow, which are subject to the path explosion problem. Inter-binary analysis is often needed because SSL/TLS API invocations may span multiple executable/library binaries, further increasing the complexity of the analysis. Second, the static analysis strategy shall be generic to handle various misuse cases in real world. We shall be able to analyze different SSL/TLS library implementations and detect different types of misuse such as incorrect certificate verification and support of deprecated SSL/TLS protocols.

To fill the gap, this paper presents SAMBA, an automated tool, which is efficient and generic for detecting SSL/TLS API misuses in Linux-based IoT binaries. SAMBA features two novel designs. First, we introduce a SSL/TLS API-centric graph (SAG), which is an inter-binary and inter-procedural API-centric control flow graph (CFG) based on the SSL/TLS APIs relevant to misuses. Second, we propose a formal expression of the SSL/TLS API misuse signatures, which are based on the APIs of interest, their parameters and return values. Therefore, based on the constructed SAG, SAMBA identifies and checks all the control flows and data flows related with APIs used in the signatures. A misuse is detected if there is a match with any signature. SAG helps avoid the heavyweight static analysis of the whole binary.

Our major contributions can be summarized as follows. We propose SAMBA, the first IoT binary analysis system for detecting two types of SSL/TLS API misuses, i.e., API call sequence misuse and API data misuse, either of which may lead to the two vulnerabilities including incorrect certificate verification and/or support of deprecated SSL/TLS protocols. We introduce two novel designs into SAMBA to make the binary analysis effective, efficient, and generic. First, SAMBA involves a three-level reduction design to construct the SAG that not only significantly reduces the size of the generated inter-procedural control flow graph but also captures all SSL/TLS API call sequences. Second, we propose a formal expression to define misuse signatures and employs a signature-based approach to effectively and efficiently detect SSL/TLS API misuse.

We implement a prototype of SAMBA that supports three architectures (ARM, MIPS, and x86), two popular SSL/TLS libraries (OpenSSL [12] and GnuTLS [13]), two types of SSL/TLS API misuse and 63 misuse signatures. SAMBA is

used to successfully analyze 115 binaries and reports all of those binaries are subject to at least one SSL/TLS misuse. QEMU is also used to validate the detected vulnerabilities of IoT firmware. SAMBA can be extended to other SSL/TLS implementations and new types of SSL/TLS misuse through our generic signature construction mechanism.

Responsible disclosure: We have reported all confirmed findings to relevant parties.

II. BACKGROUND

This section introduces the SSL/TLS protocol and two types of vulnerabilities due to SSL/TLS API misuses.

A. SSL/TLS Overview

The SSL/TLS protocol is designed to provide secure end-to-end communication and is widely used in applications and protocols such as HTTPS [14] and SMTPS [15]. Establishing an SSL/TLS connection between two parties involves multiple rounds of interactions. To simplify the use of SSL/TLS, core functionalities of the protocol are encapsulated by various third-party libraries, such as OpenSSL [12] and GnuTLS [13]. With the help of SSL/TLS libraries, developers can easily establish SSL/TLS connections in their applications.

Fig. 1 gives an example of how to establish a secure SSL/TLS connection with the OpenSSL library. Firstly, the client configures the supported SSL/TLS protocol as TLS 1.2 (a secure version) by using `TLSv1_2_client_method()` Next, a socket is set up for the SSL/TLS connection and the SSL/TLS handshake process is conducted by calling `socket(·)`, `SSL_set_fd(·)`, and `SSL_connect(·)`. To establish a secure SSL/TLS connection, it is important for the client to check the authenticity of the SSL/TLS server. To this end, the client should request the server certificate with `SSL_get_peer_certificate(·)`. If the certificate is sent by the server, its return value should not be *NULL* and `SSL_get_verify_result(·)` should be used to verify the server certificate. The verification result of `SSL_get_verify_result(·)` is stored in the *RAX* register. If the result is `0`, i.e., `X509_V_OK`, it indicates that the server certificate has passed the verification. In this case, a secure SSL/TLS connection has been established between the client and the server. Otherwise, the SSL/TLS connection is shut down as the certificate verification fails.

B. SSL/TLS API Misuse

Given the inherent complexity of the SSL/TLS protocol, diversity of SSL/TLS library implementations and inadequate documentation of some libraries, developers may inadvertently misuse the APIs, resulting in severe security risks in real-world applications [1], [2]. The most common vulnerabilities caused by SSL/TLS API misuses are support of deprecated protocols and insecure certificate verification.

Deprecated Protocol Support (DPS). As shown in Fig. 1, the first thing for a client to establish an SSL/TLS connection is to send a hello message that specifies the supported

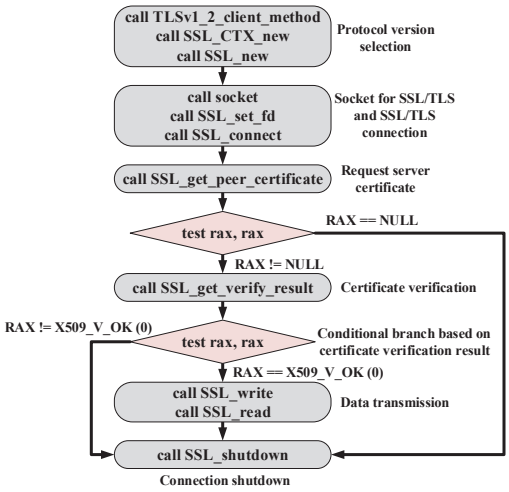


Fig. 1. Example OpenSSL API Usage.

SSL/TLS protocols. Existing research [3], [16] and RFC documents [17], [18], [19] have found certain SSL/TLS protocols susceptible to design flaws and should not be used. For example, if a client uses `SSLv3_client_method()` instead of `TLSv1_2_client_method()` to configure the support for SSL 3.0, the connection may be vulnerable to the POODLE attack [3]. To ensure a secure SSL/TLS connection, the client should avoid supporting deprecated/insecure protocols, i.e., SSL 2.0/3.0 and TLS 1.0/1.1.

Insecure Certificate Verification (ICV). During the establishment of an SSL/TLS connection, the client relies on the server certificate to ensure the authenticity and trustworthiness of the server. To this end, the client should request the certificate from the server at the handshake stage and then perform a series of checks to verify the validity of the server certificate. Otherwise, the connection is vulnerable to the MITM attack. An example of such vulnerability is that the result of `SSL_get_verify_result(·)` in Fig. 1 is properly examined.

III. SSL/TLS API MISUSES AND CHALLENGES

In this section, we first introduce two types of SSL/TLS API misuse. Then we present the technical difficulties associated with detecting SSL/TLS API misuses. Finally, we present the overview of our detection system—SAMBA.

A. Misuse Types

By analyzing documents of OpenSSL [12] and GnuTLS [13], the following two types of SSL/TLS API misuse are the API call sequence misuse (T-I) and the API data misuse (T-II), which may lead to incorrect certificate verification (ICV) and/or deprecated SSL/TLS protocol support (DPS). It can be observed that both types of misuse occur during the establishment of an SSL/TLS API connection.

1) T-I: API Call Sequence Misuse: This type of misuse occurs when a developer fails to invoke necessary APIs or invokes old APIs. For example, Fig. 1 shows a correct certificate verification API use. If a developer does not call `SSL_get_verify_result(·)`, the SSL/TLS connection

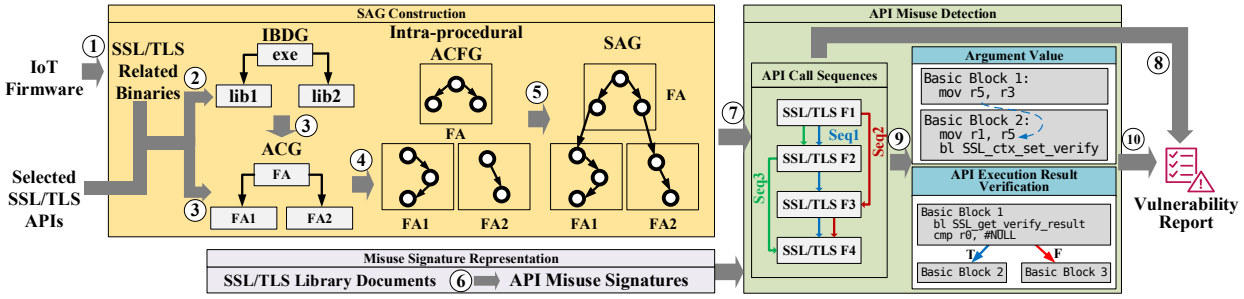


Fig. 2. System Overview

can still be established without verifying the server certificate, leading to the ICV vulnerability, which can be exploited by the MITM attack.

2) T-II: API Data Misuse.: Even if the API call sequence is correct, the misuse of data in the API calls in the sequence may still affect the security of the SSL/TLS connections. The API data misuse can be further classified into two sub-types: API argument misuse and API execution result misuse.

T-II.a: API Argument Misuse. Passing incorrect arguments to some SSL/TLS library APIs can result in insecure connections. For example, in an OpenSSL-based SSL/TLS client application, if the macro value `SSL_VERIFY_NONE` passed to the "mode" parameter of `SSL_CTX_set_verify(·, int mode, ·)`, the SSL/TLS connection can be established regardless of the server certificate verification result, causing the ICV vulnerability.

T-II.b: API Execution Result Misuse. It is crucial to check the execution results of certain SSL/TLS APIs. Otherwise, it can lead to the ICV vulnerability. These function execution results may be returned via a return value or pointer argument. As shown in Fig. 1, an OpenSSL-based SSL/TLS application uses `SSL_get_peer_certificate(·)` and `SSL_get_verify_result(·)` to request and verify the server certificate. The certificate verification result is stored in the return value of `SSL_get_verify_result(·)`. If the execution result is not correctly verified the established connection can be vulnerable.

B. Challenges

The main challenges in detecting misuses of SSL/TLS APIs in IoT are listed as follows.

(C-I) Path-explosion: In analysis of API usage in an IoT binary, extracting the SSL/TLS API call sequences is daunted by the path-explosion problem due to the complexity of the inter-procedure control flow graph (ICFG) of the binary. We find the SSL/TLS library APIs can be called in another library, and the executable indirectly calls the SSL/TLS APIs. In this case, the SSL/TLS API usage cannot be discovered by only analyzing the executable. The inter-binary analysis should be conducted and this makes the path-explosion problem worse.

(C-II) Diversity of SSL/TLS implementations: To facilitate the implementation of SSL/TLS in programming, a wide range of open-source SSL/TLS libraries have been developed, such as OpenSSL and GnuTLS. An SSL/TLS library may also consist of multiple versions. Within various SSL/TLS libraries and versions, there exists a wide variety of SSL/TLS API misuse patterns. Expressing these patterns in a general way presents a significant challenge.

IV. System Design

In this section, we first present an overview of SAMBA and then present the details of the workflow of SAMBA.

A. Overview

Fig. 2 illustrates the workflow of SAMBA. The inputs are the *IoT firmware* and *SSL/TLS APIs of interest*. In Step ① in Fig. 2, binwalk [20] can be used to extract files from the IoT firmware, which contains executables, libraries, and other files. The executables and libraries are in the binary format. We are only interested in the executable that establishes SSL/TLS connections and its dependent libraries. SSL/TLS APIs of interest refer to those SSL/TLS APIs that are used to establish an SSL/TLS connection and transfer data with the SSL/TLS server. Those APIs are related with the two misuses introduced in Sec. III-A and will be used to tackle the path explosion problem. SAMBA outputs the vulnerability report generated by our static analysis, including SAG construction, misuse signature representation and API misuse detection.

SAG construction: By analyzing the ELF header of the executable, we can tell if the executable directly calls the APIs of the SSL/TLS library or it calls wrapper functions in other libraries. In the former case, we analyze the executable to discover misuses. In the latter case, we construct an inter-binary dependency graph (IBDG) in Step ② to discover the invocation relationship between the executable and its dependent libraries that call SSL/TLS APIs for further function-level analysis. The IBDG does not contain libraries that do not call SSL/TLS APIs or their wrapper functions. The former case can be viewed as a special latter case in which the IBDG contains only the executable as a node. Next, we use the SSL/TLS APIs of interest as leaf nodes to construct API call graphs (ACGs) backward for each binary in the IBDG in Step ③. Please note whenever disassembly of the binary is needed, we use IDAPython [21]. For each function in an ACG, we then create its control flow graph, but remove those control flows not related with SSL/TLS APIs so as to create SSL/TLS API-centric control flow graph (ACFG) in Step ④. Finally we obtain the SAG by merging these ACFGs based on API invocation relationship in Step ⑤.

Misuse signature representation: To detect the API misuse introduced in Sec. III-A, we propose a formal expression of

the SSL/TLS API misuse signatures based on SSL/TLS library documents and enumerate all SSL/TLS API misuse signatures in Step ⑥. These signatures are fed into the SSL/TLS API misuse detection in Fig. 2.

API misuse detection: We first extract the API call sequences from the SAG in Step ⑦ and map API call sequences to the misuse signature so as to detect the T-I—API call sequence misuses in Step ⑧. Data flow analysis is needed for specific SSL/TLS APIs to detect T-II—API data misuses. If an API call sequence uses such an API, we find the caller of the API and perform backward taint analysis and forward taint analysis on the CFG of the caller function in Step ⑨ and check if data is properly used so as to detect T-II misuses in Step ⑩.

B. SAG *Construction*

To solve the path explosion problem, we adopt a three-level reduction method to construct the SAG for further API misuse detection, including binary-level reduction, function-level reduction, and basic-block-level reduction.

1) ***Binary-level Reduction in Step ②****:* Since some libraries call the APIs of the SSL/TLS library and indirectly provide the capability of establishing SSL/TLS connections, we build the *inter-binary dependency graph* for the target executable and its dependent libraries to represent the dependencies among them. An IBDG is a directed graph in which the root node is the executable, and other nodes represent its dependent libraries. A directed edge in the IBDG means that the source binary depends on the destination binary, i.e., the source executable/library binary call the APIs of the destination library binary.

To create the IBDG, we recursively analyze the ELF header of the executable and its dependent libraries to derive the imported functions of the executable as well as the imported and exported functions of its libraries. By correlating the imported and exported functions, we can build a raw IBDG. We then prune nodes and edges not related with SSL/TLS APIs from the raw IBDG and derive the final IBDG so as to reduce the analysis space.

2) ***Function-level Reduction in Step ③****:* We construct the call graph (CG) for each binary in the IBDG and then leverage the SSL/TLS APIs of interest to prune nodes and edges corresponding to irrelevant functions on the CG so as to reduce the graph size at the function level and derive an API-centric call graph (ACG). If the executable directly calls the SSL/TLS APIs to establish the SSL/TLS connection, we only build an intra-binary ACG of the binary to extract the function call relationships among SSL/TLS APIs of interest and the functions that call these SSL/TLS APIs. If the SSL/TLS library is indirectly dependent by an executable binary, i.e., wrapped in other libraries, we first build intra-binary ACGs for the executable and all of its libraries in the IBDG. We can then derive an inter-binary ACG by merging these intra-binary ACGs based on the function calls.

Intra-binary ACG Construction. The construction of intra-binary ACG starts with constructing a CG of the target binary. We identify all used APIs of interest as leaf nodes of the ACG by scanning the functions in the CG. Next, starting from these leaf nodes, we recursively perform backward traversal on the CG using the breadth-first search (BFS) algorithm to enumerate all parent nodes, i.e. caller functions, in order to construct the intra-binary ACG. Once we reach the root nodes (i.e., the functions that are not called by other functions), the intra-binary ACG is completely constructed. In this way, we eliminate all nodes not related with SSL/TLS APIs of interest from the original CG of the binary.

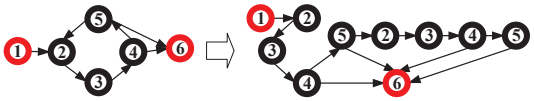


Fig. 3. Loop unrolling example

Inter-binary ACG Construction. Since circular dependencies (i.e., cycles) may exist within the IBDG and hinder the direct traversal of CGs of the binaries for inter-binary ACG construction, we discuss the ACG construction methods for the IBDG in the absence and presence of cycles respectively. In the case of an acyclic IBDG, the intra-binary ACG can be constructed backward and recursively based on the following rule: we build the intra-binary ACG for a binary only if all ACGs of its dependent libraries have already been constructed or if its successor is the SSL/TLS library. Once the intra-binary ACG is constructed and the binary is a library, we define the root nodes of the ACG (i.e., the export functions that call the the SSL/TLS APIs of interest) as the new APIs of interest. Next, we recursively traverse the IBDG backward to build the intra-binary ACGs based on the new APIs of interest until the ACG of the root node is constructed. In this way, we construct all the ACGs for the binaries in the IBDG.

We break the cycles in our context as follows. We first leverage the depth-first search (DFS) starting from the root node, i.e., the executable, of the IBDG to identify its cycle. We then take each node in the cycle as a root node to generate sub-IBDGs without other nodes in the cycle. The sub-IBDGs have two types: sub-graphs containing the SSL/TLS library and sub-graphs not containing the SSL/TLS library, in which the dependency relationship between the binaries in the IBDG and the SSL/TLS library is provided by the dependency relationship between other binaries in the cycle and the SSL/TLS library. For the sub-IBDGs containing the SSL/TLS library, we recursively build the ACG backward. After we construct the ACGs for all binaries on the first type of sub-IBDG, we can identify the APIs that are exports of the root node of the sub-IBDG and are related to the SSL/TLS API of interest, denoted as *propagation APIs*. We then find binaries in the original cycle that call the *propagation APIs* of the root node on the first type of sub-IBDG and restore the edges between the binaries that call the propagation APIs and the node that exports the propagation APIs. Next, we build the ACGs until all the binaries in the cycle have been analyzed. In this way, only the edges related with SSL/TLS APIs of interest are reserved and cycles in our context are removed.

After breaking cycles, we take the *propagated APIs* as leaf nodes to build the intra-binary ACGs for all binaries in the IBDG. Finally we merge all intra-binary ACGs to form

the inter-binary ACG for further basic-block-level analysis. It can be observed that during the ACG construction, we prune functions not related with SSL/TLS APIs of interest in the CGs of the binaries in the IBDG to reduce the graph size at the function level.

*3) **Basic block-level Reduction in Step ④**:* Since the ACG only illustrates the function level SSL/TLS dependency, we leverage the intra-procedural Control Flow Graph (CFG) of each function in the ACG to discover the SSL/TLS function call sequence. However, CFGs contain numerous basic blocks as nodes and control flow edges. This significantly increases the graph size and leads to path explosion. There may also exist loops in a CFG. The API-centric Control Flow Graph—ACFG—is derived by eliminating basic blocks not related with SSL/TLS APIs of interest and corresponding edges from a loop-free CFG and its size is reduced. Edges are added from the preceding blocks of a deleted basic block to the succeeding blocks of the deleted block.

We address loops in CFGs that introduce complexities in extracting API call sequences as follows. We first discover the loop and detect the back edge in the loop with the DFS algorithm from the root node of the CFG. For example, in Fig. 3, we perform the DFS from node 1, and discover a loop including nodes 2 to 5. The edge from node 5 to node 2 is a back edge. We create two copies of the loop. We then select one and delete its back edge. The back edge of the other loop copy is modified to point to the counterpart node in the previous loop as shown in Fig. 3 so as to construct a new loop-free graph and replace the original loop. Finally, based on the original loop, we establish edges between nodes in the loop-free graph, the entry and exit nodes, e.g., the edges from node 1 and the edges to node 6. In this way, we can construct the loop-free CFG. Unrolling the loop just once is sufficient, as further unrolling operations would result in the redundant calling of specific SSL/TLS API(s), and our document analysis indicates it does not adversely affect the security of SSL/TLS API usage.

*4) **SAG Construction in Step ⑤**:* We construct the final SAG by merging the previously constructed ACFGs. Specifically, we add edges from the nodes (i.e., basic blocks) within one ACFG that contain the instruction calling a function in the ACG to the root node in the corresponding ACFG of the called function.

C. Misuse Signature Representation

We present a formal expression of the SSL/TLS API misuse signatures and can enumerate all SSL/TLS API misuse signatures of different misuse types in various SSL/TLS implementations, addressing C-II. Fig. 4 shows the formal representation of the SSL/TLS API misuse signatures. A signature can consist of multiple Calls and Asserts that can establish an SSL/TLS connection. The Calls are SSL/TLS API calls. The args represent one or more arguments that can be passed to the APIs in the signature. The Asserts are data verification and the data can be divided into two kinds, i.e., the argument and the function execution result. The data is used in comparing with Integer, String, Boolean, and NULL in an expression.

Signature ::= ***Call***$^{+}$ | ***Assert***$^{+}$
Call ::= ***f(args)*** | ***ret = f(args)***
args ::= ***arg*** | ***arg, args***
Assert ::= ***assert(expr)***
expr ::= ***arg comp operand*** | ***ret comp operand***
operand ::= ***Integers*** | ***Strings*** | ***Boolean*** | ***NULL***
comp ::= ***==*** | ***!=***
f ::= ***target SSL/TLS APIs***

Fig. 4. SSL/TLS API misuse signature formal expression

SSL/TLS API misuse signatures can be divided into three categories. The first category includes only T-I misuses, the second includes only T-II misuses, and the third includes both T-I and T-II misuses. For each API misuse signature, it may include the misuses leading to incorrect certificate verification, deprecated protocol support or both vulnerabilities. We have defined 23 signatures for T-I misuses, 18 signatures for T-II misuses, and 22 signatures for both T-I and T-II misuses for OpenSSL and GnuTLS. Because of the page limit, please refer to [22] for detailed signatures.

Fig. 5(a) shows an example of the third category of misuse signature used to discover a T-I misuse (API call sequence misuse). The upper block of Fig. 5(a) is the block of application code in question. The bottom block is the signature, which is used to detect the misuse in the block of application code. Specifically, `SSL_get_peer_certificate(·)` is called to request the server certificate, the `SSL_get_verify_result(·)` is not called to obtain the certificate verification result, and thus the server certificate is not correctly verified. This indicates a T-I misuse.

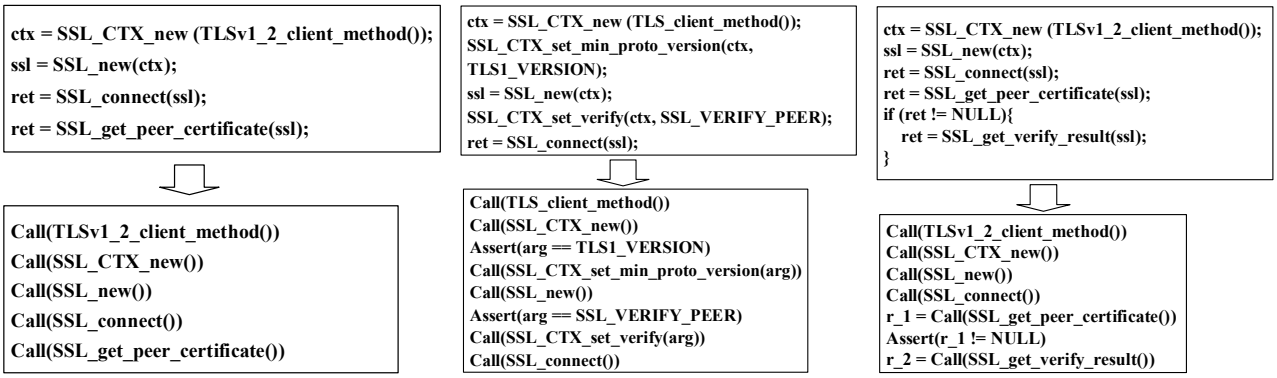
D. Misuse Detection

With defined misuse signatures in Sec. IV-C, we can perform misuse detection. We first extract SSL/TLS API call sequences from the constructed SAG by traversing the SAG forward from the root node. We filter out the sequences that do not include necessary APIs such as handshake and I/O APIs for SSL/TLS connection establishment and data transmission since the SSL/TLS connection cannot be actually established with this API call sequence. We then detect T-I and T-II misuses in the binary based on its extracted API call sequences.

1) T-I Misuse Detection: We detect the T-I misuse by comparing the extracted API call sequences with predefined misuse signatures. When a match is discovered, we identify a T-I API misuse within the binary. For example, if an extracted call sequence calls `SSL_get_peer_certificate(·)` and misses calling the `SSL_get_verify_result(·)`, it can match the signature in Fig. 5(a).

2) T-II Misuse Detection: Since analysis of API call sequences alone is insufficient to identify the T-II misuse, we perform backward and forward taint analysis based on the API call sequences to detect two types of data misuse, i.e., T-II.a—API arguments misuse and T-II.b—API execution result verification misuse.

To detect a T-II.a misuse, we identify the SSL/TLS APIs prone to argument misuse in a call sequence. We then construct the intra-procedural CFGs for the caller functions of the



(a) Signature example 1 (T-I) (b) Signature example 2 (T-II.a) (c) Signature example 3 (T-II.b)

Fig. 5. SSL/TLS API misuse signature examples

identified APIs prone to argument misuse, and locate the instructions that call the APIs. Therefore, the register that stores the argument value is used as the taint source. In our context, the argument value passed to the APIs prone to argument misuse are either integer or string. We designate the assignment instructions (e.g., "mov" and "lea" in the x86 instruction set) in which the right-hand operand is an integer or a string address as taint sinks. Now we perform backward taint analysis to trace the taint propagation from the taint source to sinks. Since we trace backward, the taint information is propagated from the left-hand operand to the right-hand operand. The analysis stops when it encounters one of the taint sinks or reaches the first instruction in the CFG. If the taint source reaches the taint sink, we can determine that the right-hand operand of the sink instruction is the value passed to the parameter. By comparing the API call sequences and the identified argument value with each signature, we can detect if the target binary contains the T-II.a misuse.

For example, `SSL_CTX_set_min_proto_version(·, version)` can configure the minimum protocol support by the binary with the value passed to the "version" parameter. If a wrong value is passed, deprecated protocols can be supported. Therefore, if we discover this API in a call sequence, a backward taint analysis should be performed to find the argument value. If the API usage matches the T-II.a misuse signature in Fig. 5(b), we identify a T-II.a misuse.

To detect a T-II.b misuse, we first identify if SSL/TLS APIs prone to execution result misuse are called in the call sequences and then construct the intra-procedural CFGs for the caller functions of the identified APIs. In order to track the function execution result, we locate the register that stores the function execution result and use it as the taint source. We mark the compare instructions on the caller function's CFG as taint sinks. We now proceed to forward trace the taint propagation. Specifically, the taint information is propagated from the right-hand operand to the left-hand operand. The analysis continues until it encounters a taint sink or reaches the last instruction of the CFG. If the taint source reaches one of the taint sinks, it indicates that the API execution result is verified and correctly used. Otherwise, if the taint source does not reach the taint sinks, it implies that the return value is not adequately verified and misuse is detected. For example, in the upper code block in Fig. 5(c), the return value of `SSL_get_verify_result(·)` is not checked, and this matches the signature depicted. A T-II.b misuse is detected.

V. Evaluation

In this section, we first introduce the experiment setup and evaluate the effectiveness of SAMBA. We then present the results of detecting vulnerabilities in real-world IoT binaries and validation of vulnerability exploitability .

A. Experiment Setup

Our experiments answer the following research questions.

- *RQ1:* How effective is the method in detecting misuses in ground truth datasets?
- *RQ2:* How does SAMBA perform in detecting real-world SSL/TLS API misuse?
- *RQ3:* Can SAMBA detect exploitable SSL/TLS API misuse vulnerabilities?

For the evaluation, we implement a prototype of SAMBA with over 3000 lines of Python code based on IDAPython [21]. According to related work [1], [2], OpenSSL and GnuTLS are two popular SSL/TLS libraries. Therefore, we implement the prototype to support the detection of SSL/TLS API misuses related with these two libraries to demonstrate SAMBA. The inter-binary analysis is conducted on a macOS laptop (with a 2.3 GHz Dual-Core i5 CPU and 8GB memory). Other experiments are conducted on a Windows computer with a 2.4 GHz Xeon CPU and 64 GB memory.

B. Effectiveness (RQ1)

In this experiment, we use a ground-truth dataset to evaluate the effectiveness of SAMBA, including a comparison with a baseline.

Ground Truth Construction. Since there is no public dataset for SSL/TLS API misuses, we create such a dataset. To reduce the manual efforts of labeling the SSL/TLS API usage, we choose Ubuntu programs for the ground truth construction. Particularly, we select 30 Ubuntu programs that employ either the GnuTLS or OpenSSL library for establishing SSL/TLS connections to construct the ground truth dataset in two steps:

TABLE I
GROUND-TRUTH EVALUATION RESULTS (RQ1)

		Ground Truth		Results							
		P	N	Suc	TP	FP	TN	FN	Coverage	Precison	Recall
ICV	SSLint	42	24	66	42	0	24	0	100%	100%	100%
	SAMBA	42	24	48	32	0	16	0	72.73%	100%	100%
DPS[1]	SSLint[2]	50	7	-	-	-	-	-	-	-	-
	SAMBA	50	7	42	36	0	6	0	73.68%	100%	100%

[1] The SSL/TLS protocol supported in 9 SSL/TLS connections is determined during execution.
[2] Related work SSLINT does not discover the deprecated protocol support vulnerability.

(i) we locate all the SSL/TLS API usage that can establish SSL/TLS connections in the selected programs; (ii) we manually check each connection to identify if it is vulnerable to ICV (incorrect certificate verification) or DPS (deprecated protocol support) because of SSL/TLS API misuses. Since the manual examination of SSL/TLS usage is performed at the source code level, the correctness of the labeling can be guaranteed.

After about 45 man-hours of manual examination, we discover 66 SSL/TLS API sequences that establish SSL/TLS connections in these 30 programs as shown in Tab. I. For the certificate verification among the 66 connections, 24 connections have correctly verified the server certificate, while the other 42 have the certificate verification misuse issue. In terms of DPS, we find 50 connections support at least one deprecated SSL/TLS protocol and only 7 connections support secure SSL/TLS protocols. Note that we do not label the protocols supported in the other 9 connections because their supported protocols are determined during program execution, e.g., configured with command line argument.

Baseline. To the best of our knowledge, all existing SSL/TLS API misuse detectors rely on source code. We compare SAMBA with SSLINT [1], a state-of-the-art source code-level SSL/TLS API misuse detector based on static analysis. SSLINT is primarily designed to discover ICV vulnerability and does not support detection of deprecated SSL/TLS protocols. Therefore, we compare the results of certificate verification detection between them.

Since the artifact of SSLINT has not been released, we have to implement a prototype of SSLINT for comparison. In our implementation, we strictly follow the design and the implementation details in the paper [1]. SSLINT employs a two-step detection procedure. First, it generates a Program Dependence Graph (PDG) to capture both data dependencies and control dependencies within a program. Next, it utilizes a graph query language to match the signatures of correct SSL/TLS API usage. If no correct SSL/TLS API usage is found in a connection, the program is reported to have flaws in using SSL/TLS APIs. To reduce the engineering efforts, Joern and Code Property Graph Query Language (CPGQL) [23] are chosen to ease our implementation. We use Joern to construct a PDG for a target program and define the correct-use signatures of SSL/TLS APIs with the CPGQL. We can then detect ICV flaws by matching the signatures in the constructed PDG.

Before conducting the comparison experiment, we assess whether the performance of our SSLINT prototype is comparable to the performance reported in the original paper [1]. We set up the same testing environment as the one in [1] (i.e., Ubuntu 12.04) and download its reported vulnerable programs with the *apt* command. 13 programs have been successfully downloaded while the repositories of other programs are no longer active. We use our SSLINT prototype to analyze these programs. The results show that the prototype has successfully detected the SSL/TLS flaws in these programs. In fact, the prototype does not find any correct SSL/TLS API usage within these programs, which aligns with the findings reported in [1]. Based on the assessment, we believe that our prototype of SSLINT has similar performance in terms of detecting SSL/TLS API misuses compared with [1] and can be used to conduct a fair comparison with SAMBA.

Evaluation Results. We use the constructed ground truth dataset to evaluate SAMBA and compare it with SSLINT. Specifically, SAMBA is evaluated on the compiled binaries of the ground truth applications while SSLINT is evaluated on their source code. The evaluation results are presented in Tab. I. Please note that we define a SSL/TLS API misuse as positive while a correct SSL/TLS API use is a negative. It can be observed that both SSLINT and SAMBA achieve a perfect precision and recall (i.e., 100%). The coverage of SAMBA does not reach 100% for the following reasons: (i) SAMBA may fail to generate a call graph due to the bugs in IDAPython when recovering the function call relationship; (ii) SAMBA stops if a timeout (set to 8 hours in our experiments) occurs during the analysis. The timeout is caused by the path-explosion problem in extracting the API call sequences. We believe that as a binary-level SSL/TLS API misuse detector, SAMBA performs well.

C. Real-world Vulnerability Detection (RQ2)

In this experiment, we use SAMBA to uncover SSL/TLS API misuse vulnerabilities in real-world IoT binaries.

Dataset Construction. Our evaluation primarily focuses on IoT binaries. To obtain these applications, we first obtain the IoT firmware published by Karonte [24] and FirmSec [25]. We also leverage the firmware crawlers provided by Firmadyne [9] to collect more IoT binary firmware samples. In total, we collect 1,143 firmware samples. We use binwalk [20] to unpack a firmware and analyze the library dependencies of the extracted binaries to identify those executables using the OpenSSL/GnuTLS library. We successfully identify 367 executables that establish SSL/TLS connections. We remove duplicated executables that have the same MD5 value (because some vendors may use the same binary in different firmware) and get 148 unique SSL/TLS executables. Among them, we find that 135 executables directly call SSL/TLS APIs to establish the SSL/TLS connections, while the other 13 executables call the wrapper functions of SSL/TLS APIs. This demonstrates the necessity of leveraging inter-binary analysis for SSL/TLS API misuse detection. Among the 148 executables, there are 71 ARM executables and 77 MIPS executables.

Detection Results. We use SAMBA to analyze all the collected 148 executables and SAMBA successfully analyzes 115 executables (including 61 ARM executables and 54 MIPS executables). Surprisingly, we find that every analyzed executable has the SSL/TLS API misuse issues. The detailed

results are presented in Tab. II. Among the vulnerable 115 executables, 94 executables are vulnerable to ICV. 112 executables are found to be vulnerable to DPS. More specifically, there are 31 executables supporting SSL 2.0, 93 executables supporting SSL 3.0, 104 executables supporting TLS 1.0, and 55 executables supporting TLS 1.1. All these detected executables are vulnerable to MITM attacks or side-channel attacks, including POODLE [3] and Lucky 13 [26].

Analysis of Failures: For the 33 executables that SAMBA fails to analyze, we manually investigate these cases and find four causes. (i) SAMBA fails to obtain a complete call graph for 12 of them due to the limitations of IDAPython in fully extracting the call graph. SAMBA cannot discover all target APIs and extract correct API call sequences from a partial call graph. (ii) SAMBA does not find the invocation of read/write-related SSL/TLS APIs (e.g., `SSL_read` and `SSL_write`) in 8 executables. In this situation, it is impossible for SAMBA to extract a complete API call sequence for vulnerability signature matching. (iii) SAMBA crashes when analyzing 8 executables due to unknown bugs maybe in IDAPython. (iv) SAMBA encounters timeout errors for 5 executables.

Efficiency. To evaluate the efficiency of SAMBA, we measure its analysis time for all successfully analyzed IoT binaries. We find that the average time cost for SAMBA to analyze an executable is only 55.45 seconds, which is quite efficient. We further evaluate the contribution of the CFG pruning technique in improving the analysis efficiency of SAMBA. To this end, we implement a raw version of SAMBA which extracts API call sequences without CFG pruning and compare its performance with SAMBA. In the raw method, we first generate the inter-procedural CFG and then directly traverse all blocks to extract the SSL/TLS API call sequences. The results show that the raw version of SAMBA cannot extract the API call sequences of 67 executables within 8 hours. Compared with the raw version, the average execution time of SAMBA is significantly reduced by over 90.8%. This result clearly demonstrates the necessity of the CFG pruning technique.

D. Validation of Vulnerability Exploitability (RQ3)

In this experiment, we validate the exploitability of the detected vulnerabilities and measure the precision of SAMBA.

Validation Methodology. We dynamically execute the vulnerable executables and set up a MITM proxy to monitor the SSL/TLS connections between the binary and the server. When we observe an SSL/TLS connection is established, we use the following two methods to confirm an ICV vulnerability and a DPS vulnerability respectively.

- To confirm an ICV vulnerability, we use the proxy to replace the original server certificate with a self-signed certificate. If the SSL/TLS connection can be established successfully with the self-signed certificate, it indicates that the binary is vulnerable to an ICV vulnerability.
- To confirm a DPS vulnerability, we set up an SSL 2.0 server, an SSL 3.0 server, a TLS 1.0 server and a TLS 1.1 server and then redirect the SSL/TLS connection to these servers. If a connection can be successfully established with one of these servers, it indicates that the tested binary supports the deprecated protocol and the connection is vulnerable to various known attacks, e.g., PODDLE [3] and Lucky 13 [26].

TABLE II
SSL/TLS API MISUSE DETECTION RESULTS FOR REAL-WORLD FIRMWARE IMAGES (RQ2)

	# of Firmware	# of Unique executables	# of Failed executables	# of ICV executables	# of DPS executables
ARM	105	71	10	56	58
MIPS	57	77	23	38	54
Total	162	148	33	94	112

TABLE III
SSL/TLS API MISUSE VALIDATION RESULTS (RQ3)

	# of SSL/TLS established executables	ICV	DPS			
			SSL 2.0	SSL 3.0	TLS 1.0	TLS 1.1
ARM	20	20	2	14	19	9
MIPS	17	17	11	16	15	3
Total	37	37	13	30	34	12

We use QEMU [27] to run the found venerable binaries via emulation. Due to the dependencies on the underlying hardware components and environments, we try our best to set up the correct hardware configuration for the emulated application. For the proxy, we use mitmproxy [28], an open-source interactive proxy that supports SSL/TLS protocols.

Validation Results. Among the 115 vulnerable IoT binary executables detected by SAMBA in our experiments, we successfully configure 37 binaries (20 ARM binaries and 17 MIPS binaries) that can be emulated with QEMU and establish the SSL/TLS connections. The other binaries failed due to the limitation of QEMU or cannot trigger the SSL/TLS connection establishment. The vulnerability validation results are presented in Tab. III. For the 37 binaries that we have successfully emulated, SAMBA reports ICV for all of them and reports DPS for 36 of them. All of the 37 binaries have been confirmed to be vulnerable to the MITM attack due to ICV. For the 36 binaries vulnerable to DPS, we successfully connect them to a server that runs a deprecated protocol. Specifically, we find 13, 30, 34, and 12 binaries that can connect to a SSL 2.0, a SSL 3.0, a TLS 1.0, and a TLS 1.1 server respectively. The results show that the SSL/TLS API misuse vulnerabilities detected by SAMBA are indeed exploitable and SAMBA can detect them with a precision of 100%.

VI. DISCUSSION

In this section, we discuss limitations of SAMBA and advantages of binary analysis in detecting SSL/TLS API misuse.

A. Limitations

This paper acknowledges two limitations. Firstly, our system conducts static analysis. Therefore, during data flow analysis, if the argument value passed to the SSL/TLS API prone to argument misuse is determined during binary application execution, e.g., being configured with a command line argument. It cannot be detected using a static analysis system. Secondly, our data flow analysis is an intra-procedure analysis. If the argument is assigned and the execution result is verified outside

the caller function of the SSL/TLS APIs. To address this issue, an inter-procedure data analysis is required and our system may generate inaccurate reports. However, while constructing our ground truth dataset, we notice that the argument value and the API execution result verification are closely associated with the SSL/TLS API prone to misuse. Thus, in our context, intra-procedure data flow analysis suffices.

B. Advantages of Binary-level Analysis

With the evolution of the SSL/TLS libraries, some APIs have been deprecated and replaced with new ones to support similar functionalities. However, for the consideration of compatibility, some deprecated APIs are renamed to the corresponding new APIs by preprocessor macros instead of directly removed. For example, the `SSLv23_client_method()` in OpenSSL is used to configure the client application to support SSL 2.0 and later protocols, and it has been deprecated after OpenSSL 1.1.0. If an application calls `SSLv23_client_method()` and is compiled with OpenSSL 1.1.0, the API call is replaced with `TLS_client_method()` instead. In this situation, the SSL/TLS protocols supported by the application are SSL3.0 and later protocols. Such practice illustrates the advantages of binary analysis in detecting SSL/TLS misuses. That is, the source code based methods [1], [2] will report more false positives than binary-level analysis tools due to the usage of preprocessor macros in SSL/TLS libraries.

VII. Related Work

In this section, we present the most related work on API misuse vulnerability detection and static analysis based vulnerability detection methods.

A. API Misuse Vulnerability Detection

Third-party libraries play an important role in simplifying program development and the issue of API misuse attracts increasing attention [29], [30]. To address the API misuses, several detection methods [1], [2], [7], [31], [32] have been proposed. These methods can be divided into two categories: source-code-based misuse detection [1], [2], [31] and binary-code-based misuse detection [7], [32]. The ource-code-based methods require access to source code of the target SSL/TLS application, and this limits their usage when the source code of the application and/or its dependent libraries is not available. The binary-code-based methods are more general since only binary files are required. Previous works focus on detecting cryptographic misuses [7], [32], mainly target the arguments passed to cryptographic APIs, and analyze if a specific API is used and if the argument passed to the API violates the correct usage rules. However, by analyzing documents of SSL/TLS libraries, we find the execution result of some APIs should be examined to verify the certificate authenticity. SAMBA can address the deficiencies of related work.

B. Vulnerability Detection with Static Analysis

Software analysis can be either static or dynamic analysis. In this paper, we focus on analyzing the IoT applications. Due to the diverse instruction sets and peripherals used in IoT devices, it is difficult to dynamically execute binaries even with an emulator such as QEMU. Therefore, we adopt static analysis methods to discover SSL/TLS API misuse vulnerabilities. There is related work using static analysis to discover IoT application vulnerabilities [33], [24], [34], [35], For example, Shoshitaishvili et al. [33] present a system to automatically discover authentication bypass vulnerabilities in binaries based on backward slicing and symbolic execution. However, our work is the first to detect SSL/TLS API misuses in IoT firmware.

VIII. Conclusion

This paper introduces SAMBA, the first automatic tool designed to detect misuses of SSL/TLS APIs in IoT binaries. SAMBA utilizes a three-level reduction technique to construct an SAG for API call sequence extraction. Backward and forward taint analyses are then applied based on these sequences to understand the data flows of APIs prone to data misuse. By formulating API misuse signatures for OpenSSL and GnuTLS, SAMBA matches these signatures at the control and data flow levels to identify SSL/TLS API misuses in IoT binaries. Extensive experiments are conducted to validate SAMBA. We evaluate SAMBA with 30 Ubuntu SSL/TLS binaries, are able to analyze 73.21% of those binaries and achieve a precision of 100%. We also use SAMBA and successfully analyze 115 IoT binaries. We find 94 of the 115 IoT binaries are vulnerable to incorrect certificate verification, and 112 out of them support deprecated protocols. To validate the identified vulnerabilities, we emulate 37 IoT binaries with QEMU, and confirm the vulnerabilities are indeed present in these binaries. Particularly, we find vulnerabilities related to certificate verification in all the 37 binaries and 36 binaries support deprecated protocols. Our experiment results highlight the effectiveness and efficiency of SAMBA, shedding light on the insecure use of SSL/TLS APIs in today's IoT devices.

Acknowledgement

This research was supported in part by National Key R&D Program of China No. 2023YFC3605804, by National Natural Science Foundation of China Grant Nos. 62072103, 62022024, 61972088, and 62232004, by US National Science Foundation (NSF) Awards 1931871, 1915780, and US Department of Energy (DOE) Award DE-EE0009152, Jiangsu Provincial Key R&D Programs Nos. BE2021729, BE2022680, BE2022065-5, Jiangsu Provincial Natural Science Foundation of China Grant No. BK20220806, Jiangsu Provincial Key Laboratory of Network and Information Security Grant No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China Grant No. 93K-9. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, V. N. Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting SSL usage in applications with SSLINT. In *Proceedings of the 36th Symposium on Security and Privacy (S&P'15)*, pages 519–534, San Jose, CA, USA, May 17-21 2015.

[2] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing API usages through semantic cross-checking. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*, pages 363–378, Austin, TX, USA, August 10-12 2016.

[3] Bodo Möller, Thai Duong, and Krzysztof Kotowicz. This POODLE bites: exploiting the SSL 3.0 fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf, 2014 [online].

[4] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. Smv-hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in android apps. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, pages 1–14, San Diego, CA, USA, February 23-26 2014.

[5] Yingjie Wang, Guangquan Xu, Xing Liu, Weixuan Mao, Chengxiang Si, Witold Pedrycz, and Wei Wang. Identifying vulnerabilities of SSL/TLS certificate verification in android apps with static and dynamic analysis. *Journal of Systems and Software*, 167:110609, 2020.

[6] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. AVATAR: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14)*, San Diego, California, USA, February 23-26 2014.

[7] Li Zhang, Jiongyi Chen, Wenrui Diao, Shanqing Guo, Jian Weng, and Kehuan Zhang. Cryptorex: Large-scale analysis of cryptographic misuse in iot devices. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID'19)*, pages 151–164, Chaoyang District, Beijing, China, September 23-25 2019.

[8] Bo Feng, Alejandro Mera, and Long Lu. P2IM: scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, pages 1237–1254, August 12-14 2020.

[9] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*, pages 1–16, San Diego, CA, USA, February 21-24 2016.

[10] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Proceedings of the 1st Workshop on Binary Analysis Research (collocated with NDSS Symposium) (BAR)*, San Diego, CA, USA, February 18 2018.

[11] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *Proceedings of the 29th USENIX Security Symposium (USENIX Security'20)*, pages 19–36, August 12-14 2020.

[12] OpenSSL Project. Open source SSL/TLS library . https://www.openssl.org, 1999 - 2021 [online].

[13] Tim Rühsen, Daiki Ueno, and Dmitry Baryshkov. The GnuTLS transport layer security library. https://www.gnutls.org/index.html, 2021 [online].

[14] Eric Rescorla. HTTP Over TLS. https://datatracker.ietf.org/doc/html/rfc2818, 2000 [Online].

[15] Paul Hoffman. SMTP Service Extension for Secure SMTP over Transport Layer Security. https://datatracker.ietf.org/doc/html/rfc3207, 2002 [Online].

[16] Eman Salem Alashwali and Kasper Rasmussen. What's in a downgrade? A taxonomy of downgrade attacks in the TLS protocol and application protocols using TLS. In *Proceedings of the 14th International Security and Privacy in Communication Networks Conference (SecureComm'18)*, pages 468–487, Singapore, August 8-10 2018.

[17] Tim Polk, and Sean Turner. Prohibiting Secure Sockets Layer (SSL) Version 2.0. https://datatracker.ietf.org/doc/html/rfc6176, 2011 [Online].

[18] Kathleen Moriarty, and Stephen Farrell . Deprecating TLS 1.0 and TLS 1.1. https://datatracker.ietf.org/doc/html/rfc8996, 2021 [Online].

[19] Richard Barnes, Martin Thomson, Alfredo Pironti, and Adam Langley. Deprecating Secure Sockets Layer Version 3.0. https://datatracker.ietf.org/doc/html/rfc7568, 2015 [Online].

[20] Heffner Craig. Binwalk: Firmware analysis tool. https://code.google.com/p/binwalk/, 2010 [online].

[21] Hex-Rays SA. IDAPython. https://github.com/idapython/src, 2015.

[22] SAMBA. https://github.com/kzLiu2017/SAMBA.

[23] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th Symposium on Security and Privacy (S&P'14)*, pages 590–604, Berkeley, CA, USA, May 18-21 2014.

[24] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *Proceedings of the 41st Symposium on Security and Privacy (S&P'20)*, pages 1544–1561, San Francisco, CA, USA, May 18-21 2020.

[25] Binbin Zhao, Shouling Ji, Jiacheng Xu, Yuan Tian, Qiuyang Wei, Qinying Wang, Chenyang Lyu, Xuhong Zhang, Changting Lin, Jingzheng Wu, and Raheem Beyah. A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware. In *Proceedings of the 31st International Symposium on Software Testing and Analysis (ISSTA'22), Virtual Event*, pages 442–454, South Korea, July 18 - 22 2022.

[26] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *Proceedings of the 34th Symposium on Security and Privacy (S&P'13)*, pages 526–540, Berkeley, CA, USA, May 19-22 2013.

[27] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 Annual Technical Conference (ATC'05)*, pages 41–46, Anaheim, CA, USA, April 10-15 2005.

[28] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/, 2010–. Version 7.0.

[29] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In *Proceedings of the 43rd Symposium on Security and Privacy (S&P'22)*, pages 614–631, San Francisco, CA, USA, May 22-26 2022.

[30] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P. Carloni, and Simha Sethumadhavan. CRYLOGGER: detecting crypto misuses dynamically. In *Proceedings of the 42nd Symposium on Security and Privacy (S&P'21)*, pages 1972–1989, San Francisco, CA, USA, 24-27 May 2021.

[31] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. ARBITRAR: user-guided API misuse detection. In *Proceedings of the 42nd Symposium on Security and Privacy (S&P'21)*, pages 1400–1415, San Francisco, CA, USA, May 24-27 2021.

[32] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-hunt: Pinpointing insecure cryptographic keys from execution traces. In *Proceedings of the 25th Conference on Computer and Communications Security (CCS'18)*, pages 412–425, Toronto, ON, Canada, October 15-19 2018.

[33] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*, pages 1–15, San Diego, CA, USA, February 8-11 2015.

[34] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. In *Proceedings of the 37th Symposium on Security and Privacy (S&P'16)*, pages 138–157, San Jose, CA, USA, May 22-26 2016.

[35] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *Proceeding of the 30th USENIX Security Symposium (USENIX Security'21)*, pages 303–319, August 11-13 2021.