CRYPTOREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices

Li Zhang[*], Jiongyi Chen[†], Wenrui Diao[‡§(✉)], Shanqing Guo[‡§], Jian Weng[*], and Kehuan Zhang[†]
∗Jinan University, `{zhanglikernel, cryptjweng}@gmail.com`
†The Chinese University of Hong Kong, `{cj015, khzhang}@ie.cuhk.edu.hk`
‡Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education,
Shandong University, `{diaowenrui, guoshanqing}@sdu.edu.cn`
§School of Cyber Science and Technology, Shandong University

Abstract

Cryptographic functions play a critical role in the secure transmission and storage of application data. Although most crypto functions are well-defined and carefully-implemented in standard libraries, in practice, they could be easily misused or incorrectly encapsulated due to its error-prone nature and inexperience of developers. This situation is even worse in the IoT domain, given that developers tend to sacrifice security for performance in order to suit resource-constrained IoT devices. Given the severity and the pervasiveness of such bad practice, it is crucial to raise public awareness about this issue, find the misuses and shed light on best practices.

In this paper, we design and implement CRYPTOREX, a framework to identify crypto misuse of IoT devices under diverse architectures and in a scalable manner. In particular, CRYPTOREX lifts binary code to a unified IR and performs static taint analysis across multiple executables. To aggressively capture and identify misuses of self-defined crypto APIs, CRYPTOREX dynamically updates the API list during taint analysis and automatically tracks the function arguments. Running on 521 firmware images with 165 pre-defined crypto APIs, it successfully discovered 679 crypto misuse issues in total, which on average costs only 1120 seconds per firmware. Our study shows 24.2% of firmware images violate at least one misuse rule, and most of the discovered misuses are unknown before. The misuses could result in sensitive data leakage, authentication bypass, password brute-force, etc. Our findings highlight the poor implementation and weak protection in today's IoT development.

1 Introduction

As an emerging field, the Internet of Things (IoT) system is on the track of rapid development. IoT devices have been deployed in several scenarios, such as smart home, smart transportation, and so forth. A recent marketing research report forecasts that the amount of IoT devices will grow to around 19 billion worldwide in 2022 [24].

Different from the traditional embedded devices, IoT devices (such as smart-home devices) are usually equipped with multiple sensors and connected to the network. In this context, the security of IoT devices becomes crucial as it involves not only the data privacy of users [13] but also the risk of dangerous incidents [28]. Due to the requirements of usage scenarios and the limitation of manufacturing costs, IoT devices often have customized hardware and software configurations. Such a closed hardware-software environment gives people an illusion of safety. However, that is not the case, and IoT related vulnerabilities emerge endlessly [10, 31, 38, 43, 50]. Given the prevalence of vulnerable devices, we believe these disclosed incidents are just the tip of the iceberg. What is worse, until now, there has not been a well-established set of technical standards for IoT security.

Previous studies of identifying vulnerabilities in IoT devices have been traditionally focused on memory corruptions [16–18, 21, 39], authentication bypass [46], and domain specific vulnerabilities such as BadUSB [30]. However, currently no tool can automatically identify crypto misuses in IoT devices. As a consequence, large-scale security analysis of crypto misuse in IoT devices has never been conducted before. On the one hand, existing solutions target specific platforms such as Android and iOS [23, 26, 40]. They are less suitable for IoT devices that often involve different architectures. For example, Android applications are provided in a reversible bytecode format, whereas IoT applications are compiled to machine code that could run on various CPU architectures (MIPS, ARM, PowerPC, and so forth).

On the other hand, prior works rely on specific crypto libraries and do not handle self-defined crypto functions. Given the lack of security standards in IoT development, developers tend to use self-defined crypto functions that wrap standard crypto functions. Although several well-established crypto libraries provide well-designed and carefully-implemented crypto APIs to facilitate the deployment of secure modules, there is no guarantee whether these crypto APIs are used and wrapped correctly. For example, inexperienced developers may use a non-random initialization vector for block cipher mode encryption or use static seeds for random number generation functions. Such a problem usually results in confidential data leakage and even system intrusion.

In this paper, we introduce CRYPTOREX, a framework that achieves automated and large-scale analysis of crypto misuse in IoT firmware. On a high level, we first lift the binary code with different architectures to the unified intermediate representation (IR) format. Then we recover the stack layouts to precisely identify the arguments of low-level crypto API arguments and track the definition of the arguments with taint analysis. In order to further capture the self-defined crypto functions and the misuses, CRYPTOREX maintains a list of crypto APIs that can be dynamically updated during taint analysis.

To demonstrate the feasibility of CRYPTOREX, we implemented a prototype of CRYPTOREX and carried out a large-scale experiment based on 1327 firmware images (from 12 vendors, in 7 different architectures) crawled from the Internet. The device types include IP camera, network attached storage, router, smart plug, smart bulb, and so forth. CRYPTOREX successfully unpacked 521 firmware images and identified 679 crypto misuses. Surprisingly, the experiment also demonstrates remarkable performance: on average, it takes only 1120 seconds for CRYPTOREX to complete one firmware analysis. Further investigation shows that 126 firmware images violate at least one crypto misuse rule. The misuses could result in the compromise of secrecy, authentication bypass, password brute-force, and etc.

With a full implementation and a comprehensive evaluation, CRYPTOREX makes the first step towards scalable and quantitative measurement for (in)secure crypto usage in IoT devices. We make this tool publicly available for continuous research on IoT firmware analysis.[1]

Contributions. The main contributions of this paper are:

- We designed a new analysis framework – CRYPTOREX, which can automatically identify crypto misuses in IoT devices. With new techniques such as stack layout recovery and dynamic update of crypto APIs, it can achieve reliable cross-architectural analysis in a large-scale manner.

- We performed the first large-scale measurement study on (in)secure crypto usage over a large number of firmware images. Our study has brought to light the worrisome situation (questionable practice and weak protection) in IoT development.

Roadmap. The rest of this paper is organized as follows. Section §2 gives the background knowledge of IoT firmware analysis and crypto misuse. The detailed design of CRYPTOREX is elaborated in Section §3. The evaluation results are summarized in Section §4. Section §5 discusses some limitations of our framework and experiments. Section §6 reviews the related work, and Section §7 concludes this paper.

2 Background

In this section, we provide the necessary background about firmware analysis and cryptography misuse.

2.1 Security Analysis of IoT Firmware

Firmware is a specific class of computer software that provides the low-level control for the device's specific hardware. Unlike PCs, for which software engineers develop multipurpose applications, firmware is usually designed for special purposes and runs on embedded devices (e.g., IoT devices) with limited resources and diverse architectures. Unfortunately, the firmware-specific features have also introduced several challenges to the security analysis. Here we summarize the challenges from the aspects of dynamic analysis and static analysis.

- **Dynamic analysis**: In dynamic analysis, the firmware is executed in a controlled environment. A bare-metal analysis (based on real devices) could output the most accurate result, but it needs the support of an exposed debug port on the device. Unfortunately, many manufacturers disable the debug port for security concerns. An alternative solution is to run the entire firmware or embedded programs in an isolated emulator. The challenge is the lack of non-volatile memory (NVRAM) parameters, which causes runtime failures during dynamic analysis. In previous work, Chen et al. [15] simulated the NVRAM parameters using userspace libraries. However, it is not suitable for a large-scale analysis due to the diversity of architectures.

- **Static analysis**: Compared with dynamic analysis, static analysis scrutinizes the binary code of firmware, instead of relying on emulation environments. In most cases, it achieves a balance of efficiency and accuracy. However, developing a unified static analysis framework for various IoT devices with different underlying architectures (MIPS, ARM, PowerPC, and so forth) is not a simple task. The disassembled binary files may contain different order sets with different operations and side-effects, which leads to various calling conventions and different stack layouts. Consequently, this could cause difficulty to further analysis such as recovering function arguments.

Our approach: IR-based analysis. Through the above discussion, for large-scale security analysis, the difficulty mainly comes from the non-unified underlying architectures of IoT firmware. To bridge the gap, we lift diverse binary code to

[1] https://github.com/zhanglikernel/CRYPTOREX

a unified intermediate representation (IR). In the process of compiling, the source is transformed into IR and then binary code. Vice versa, we can lift the binary code of different architectures to the same IR, and the subsequent analysis could be based on the IR. Previous work [20,46] has also demonstrated the feasibility of firmware analysis.

2.2 Cryptography Misuse

Though the standard cryptographic libraries provide well-implemented and well-defined APIs, developers may not fully understand the API documentation and misuse the APIs by delivering improper arguments, which could result in the compromise of confidentiality in network communication and data storage. In this paper, we focus on the inappropriate use of crypto functions and assume that the involved crypto algorithms are secure. Based on the study of Egele et al. [23] and Lazar et al. [33], we use the following six rules in cryptography that should be followed by IoT developers. As indicated in the OWASP guideline [44], these time-tested rules cover common misuses in symmetric key encryption, password-based encryption, and random number generation.

- **Rule 1.** *Do not use electronic code book (ECB) mode for encryption*. The ECB mode cannot provide strong enough security guarantee. For example, the `AES_ecb_encrypt` function of the `libcrypto` library should not be used for security-related modules.

- **Rule 2.** *Do not use a non-random initialization vector (IV) for ciphertext block chaining (CBC) encryption*. If the IV is static, the encryption scheme is considered insecure. For example, developers could use the `gcry_set_iv` function provided by `libgcrypt` to initialize the IV for the subsequent encryption operations.

- **Rule 3.** *Do not use constant encryption keys*. Constant encryption keys would bring the direct risk of cracking the encryption schemes. For example, when developers invoke the AES encryption function of the `wolfcrypt` library, they could use `wc_AesSetKey` to specify a secure key.

- **Rule 4.** *Do not use constant salts for password-based encryption (PBE)*. In Linux, the most frequently used API for password encryption is `char *crypt(const char *key, const char *salt)` of `libcrypt`. Almost all Linux systems use it to encrypt users' passwords and save the outputs to `/etc/shadow`. The parameter `salt` could not be assigned as a constant value.

- **Rule 5.** *Do not use fewer than 1000 iterations for PBE*. For example, in the function `Evp_BytesTokey` of `libcrypto`, the argument `count` specifies the round of iterations. Some developers may set small values for performance consideration, which would result in the risk of brute-force attacks.

- **Rule 6.** *Do not use static seeds for random number generation (RNG) functions*. When a program needs to generate secure random numbers, it should not use `rand()` or `srand()` with a constant seed.

To provide an intuition, we list some crypto misuse examples in Table 1.

2.3 Intermediate Representation

There are several available IRs designed for different purposes, such as REIL [22], LLVM [32], and BAP [12]. Since our analysis focuses on the function arguments, intending to track variable types, the IR should support static data-flow analysis and represent registers and memory locations in a unified format, even the executables have different architectures.

In our work, we employ Valgrind's VEX IR [41] as the representation format. The VEX IR and its Python bindings PyVEX [46] provides some features which fit our requirements ideally.

- *Static program slicing*: PyVEX supports static program slicing. PyVEX translates binary code to IR code and divides it into basic blocks that are in the form of IRSBs (IR Super-Block). With IRSBs, we can conveniently construct the control flow graph (CFG) and data flow graph (DFG).

- *Base operations*: After disassembly, an assembly instruction is transformed into multiple statements in VEX. A statement is an "atomic action“ which contains an operand and an IR expression. Besides, VEX classifies all operations into four base statements: Write Temp, Put Register, Store Memory, and Exit.

  PyVEX transforms binary code into statements, which are "atomic actions" defined by VEX.

3 Design of CryptoREX

Here we describe the detailed design of CryptoREX. At a high level, CryptoREX takes a raw firmware image as the input and outputs a report indicating the misuse of crypto functions in the firmware image. Mainly, the analysis procedure consists of five steps (as shown in Figure 1):

- **Firmware Acquisition and Pre-processing.** First, we develop a crawler to automatically download firmware images from IoT vendors' websites and then extract executable files from them.

- **Lifting to VEX IR.** Next, we lift the binary executables in various architectures to VEX IR. The subsequent analysis is conducted upon the unified representation.

Table 1: Examples of Crypto Misuse

Rule #	[Library]: Function	Parameter	Misuse Condition
Rule 1	[libgcrypt]: gcry_error_t gcry_cipher_open(gcry_cipher_hd_t *hd,int algo, int mode,unsigned int flag)	mode	== 1
Rule 2	[wolfcrypt]: int wc_AesSetIv(Aes *aes, const byte *iv)	iv	static string
Rule 3	[Nettle]: void aes192_set_encrypt_key (struct aes192_ctx *ctx, const uint8_t *key)	key	static string
Rule 4	[libcrypt]: char *crypt(const char *key, const char *salt)	salt	static string
Rule 5	[libcrypto]: int EVP_BytesToKey(const EVP_CIPHER *type, const EVP_CHPHER *type, const unsigned char *salt, const unsigned char *data, int datal, int count, unsigned char *key, unsigned char *iv)	count	< 1000
Rule 6	[C standard library]: void srand(int seed)	seed	static integer

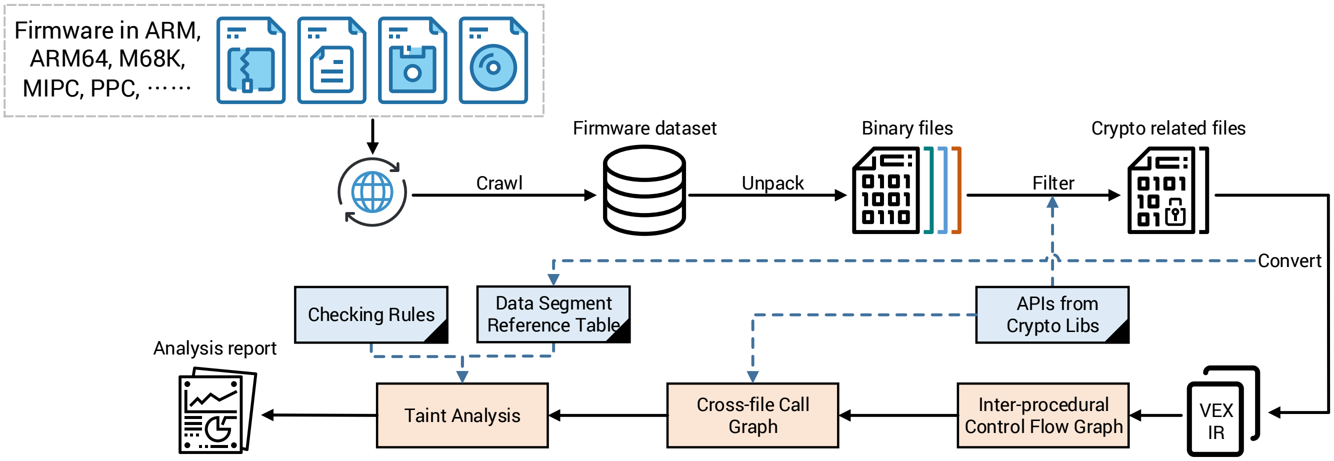

Figure 1: Framework overview.

- **Inter-procedural Control Flow Graph Construction.** Then, we construct inter-procedural control flow graphs (ICFG) for the executable files. Such graphs capture data flows across function calls.

- **Cross-file Call Graph Construction.** After that, a cross-file call graph for each crypto function is built to facilitate the data-flow tracking (in the next step) which crosses multiple executables.

- **Taint Analysis.** In the last step, we perform a backward taint analysis to track how each crypto function argument is defined. If the definition triggers the misuse rules, CRYPTOREX outputs a report listing the discovered crypto misuse cases.

3.1 Firmware Acquisition and Pre-processing

Since there is no well-established IoT firmware dataset, we developed a web crawler based on the prototype tool of Chen et al. [15] to download firmware from the IoT vendors' websites automatically. Also, the existing tool only supports downloading from FTP servers. Therefore we added a module to support firmware downloads from dynamically generated websites.

For a timely firmware update, developers tend to pack firmware images to save transmission time and storage space. As a result, most of the firmware images that we collected are compressed with various compression algorithms and cannot be directly analyzed. To tackle this problem, we use the state-of-the-art firmware unpacking tool Binwalk [2]. It integrates several file system signatures and decompression algorithms to identify file systems and extract executables (binary files) from the firmware.

3.2 Lifting to VEX IR

File filter. To reduce the time consumption of IR conversion, CRYPTOREX first filters out the binary files that do not invoke the crypto APIs. Specifically, we use Buildroot [3], a cross-compilation tool, to analyze the header information of each file to check whether cryptography libraries are included. If the cryptography libraries are not included, we then ignore the binary file. In practice, the filter utilizes the API data extracted from seven widely used open-source C/C++ cryptography libraries: `libcrypto` [42], `libcrypt` [29], `cryptlib` [4], `LibTomCrypt` [7], `libgcrypt` [6], `wolfcrypt` [9], and `Nettle` [8]. These libraries have covered nearly 100% usage cases in our firmware dataset.

Enhanced conversion. After that, built on top of the binary analysis framework Angr [1], CRYPTOREX invokes Angr APIs to disassemble binary files and lift different low-level instruction sets to the unified VEX IR. However, the direct binary-IR conversion is not sufficient to meet our requirements: (1) The call relations of Angr is incomplete because it only considers explicit invocation addresses. If the address is put into a register or memory, Angr cannot locate it. (2) Type information of variables is lost, which affects the data-flow tracking (especially the function parameters). (3) The function arguments are often passed via the register, stack, or both, and follow specific conventions. If the binary code is lifted to the IR, architecture-specific calling convention will be lost.

To solve the first two shortcomings, based on the functionalities of IDA Pro [5], we develop a recovery script to (1) locate the actual addresses of jump instructions to complete the function call relations and (2) infer data types (and save them as a *data segment reference table*) to facilitate the subsequent data-flow tracking. For the third shortcoming, we extract the arguments passing rules of different architectures in advance. During the testing, we identify the architecture types (also obtained through IDA Pro) of firmware images and apply the corresponding arguments passing rules.

3.3 Inter-procedural Control Flow Graph Construction

After that, we construct the *inter-procedural control flow graph* (ICFG), which is the foundation for the subsequent inter- and intra-procedural data-flow analysis. In particular, our ICFG construction starts with the entry point (i.e., `start()`) of each executable, and traverses functions and basic blocks through depth-first searching.

We consider function call relations in the graph, in order to support inter-procedural analysis. Except for the functions that can be reached from the entry point, we also discover isolated functions and their call relations, by scanning code segments with pre-defined function signatures [5]. In this way, function call relations and API call sites can be discovered as many as possible. This step is hugely beneficial for library files because they only contain isolated functions for external use.

Also, though Angr is able to resolve explicit target addresses, it cannot resolve implicit call relations. Therefore, to improve the precision of data-flow analysis (i.e., implicit call relations), we utilize the data segment reference table obtained from Section 3.2 and perform value-set analysis [11] to recover the addresses of indirectly invoked functions. In other words, for indirect jumps (e.g., `jr $t0` in MIPS), we simulate previous instructions and compute the actual value (or the value range) of the register or memory location. If the destination is identified as a function, we then add it to the function call relations.

Furthermore, given that loop structures often contain computation-intensive instructions (like increment operations) and have less variable definitions and uses, we flatten the loop structure so that each loop is executed only a few times during data-flow analysis. This operation can significantly reduce the time consumption of loop processing.

3.4 Cross-file Call Graph Construction

We notice that considerable IoT solution providers defined their own crypto APIs wrapping low-level crypto APIs (from other libraries), which could be treated as some kind of optimization to facilitate the internal development process. Executable files could either invoke the original low-level APIs or the self-defined APIs. This phenomenon requires us to capture the function call relations between them during the data-flow analysis. Therefore, in this step, we construct the *cross-file call graph* (CFCG for short) that involves multiple executable files and represents the chains of function calls for each low-level crypto API. With such a representation, we can dynamically update the crypto API list and further detect the misuse of self-defined crypto APIs.

In the beginning, each crypto API (from `libcrypto`, `libcrypt`, `cryptlib`, `Nettle`, `libgcrypt`, `wolfcrypt`, and `LibTomCrypt`) acts as the starting point of a call graph. Next, we construct the chains of function calls for the crypto API. To this end, we first scan the call sites of the crypto API in the extracted executables and then recursively chain the callers of the functions that invoke the crypto API. For the non-library executables, we only need to consider internal functions and imported functions. For library files, exported functions should also be considered because other executables can further invoke them.

As an example, Listing 1 is a piece of code snippet from our dataset. The `smm` program is extracted from D-Link DSR-150 (VPN router) which supports up to 65 VPN tunnels. After code review, we find its traffic encryption module is implemented through invoking the self-defined crypto API `DES_ProcessFile()` provided by library `libSys.so`. In the definition of `DES_ProcessFile()`, low-level crypto API `DES_string_to_key()` is invoked to specify the key used by the following encryption. Except for that, both `DES_string_to_key()` and `DES_ProcessFile()` can also be invoked by other executables. We show the result of CFCG (partial) in Figure 2.

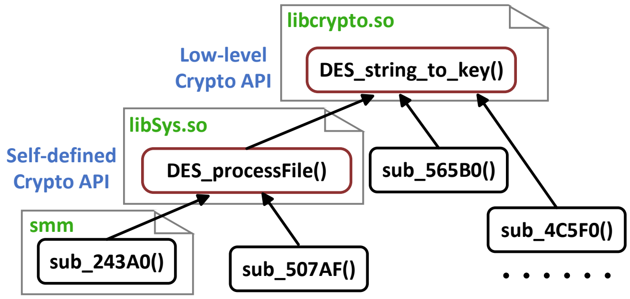

Figure 2: Example: cross-file call graph of Listing 1.

```
/***** libSys.so *****/
signed int __fastcall DES_processFile(int
    a1, const char *a2, const char *a3,
    int a4){
  int v7;
  v7 = a4;
  ...
  // low-level crypto API
  DES_string_to_key(v7,&v18);
  ...
  while(fread(&ptr,&v15,&v14,v5)){
    DES_ecb_encrypt(&ptr,&v15,v14,v5)
  }
}
//***** smm *****/
signed int __fastcall sub_243A0(char *a1,
    int a2){
  v2 = a2;
  v3 = a1;
  if(v3){
    v8 = v3;
  }else{
    v8 = "/pfrm2.0/sslvpn/var/conf/smm.
        conf";
  }
  if(v2)
    // defined in libSys.so
    DES_processFile(1,"tmp/.smm.clr",v8,"
        root123");
}
```

Listing 1: Self-defined crypto API: D-Link DSR-150.

3.5 Taint Analysis

Our final purpose is to detect whether the crypto APIs are misused in IoT firmware. We perform a static taint analysis (backward tracking) to tag the function arguments at the call sites and determine their actual values for checking. To this end, the first step is to precisely identify the sources, which are the arguments of crypto APIs. Then, we identify the wrappers of crypto APIs and dynamically update the API list based on how the function arguments propagate. In the end, the misuse rules are applied at the sinks.

Taint sources. Although we have the prototypes of crypto APIs (as shown in Table 1), the arguments in binary code are not matched with the defined parameters. Therefore, to tag the taint sources (i.e., the arguments of crypto APIs), we need to identify which function argument corresponds to the function parameter that may lead to crypto misuse. For that, we utilize the calling conventions: the arguments passing rule of the firmware under testing has been matched and recorded in the step of lifting to VEX IR (Section §3.2).

In the procedure, a register is tagged as a taint source when a function argument is passed with it. For stack-based passing, we need to recover the stack layout at the call sites statically, in order to determine which stack variable serves as a function argument. For this reason, we first compute the values of the stack pointer and the base pointer to determine the range of the caller's stack frame. Then we locate the stack-based arguments at the memory locations whose addresses are specified by offsetting the caller's stack pointer.

Taint propagation. To build data dependence, we employ the use-define chain algorithm of Angr. However, it incurs false negatives at array operation APIs of C libraries. For example, the function `strcpy(dest, src)` copies the string from the source address to the destination address. However, the data dependency is not built between these two variables. To solve the problem, we implemented a module to simulate the functionality of array operation APIs (e.g., `memset()` and `memcpy()`) and build data dependency between related variables. In the meantime, we also *dynamically update the list of crypto APIs* during backward taint analysis. On the CFCGs that we construct, we add a function (i.e., a self-defined crypto API) to the list of crypto APIs if one of its function parameters is passed to the crypto APIs as an argument. As an example, the function `DES_ProcessFile()` is a self-defined crypto API (shown in Figure 2).

Taint sinks. We define taint sinks at constants. A constant can be interpreted as either a pointer or an immediate value, based on the types of function arguments given in the function prototypes. If it is a pointer, we get its referenced data in the data segment reference table generated in Section §3.2. If the constant is an immediate value, no further step is required. After the interpretation, we analyze whether the data is matched with the specifications defined in misuse rules. If a misuse condition is triggered, such a case is considered as violation and recorded in the analysis report.

4 Implementations and Evaluation

In this section, we give the implementation details of CRYPTOREX and the evaluation results based on a large-scale real-world IoT firmware dataset.

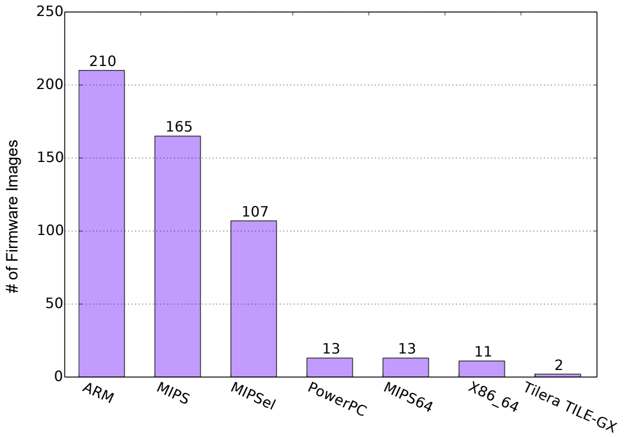

Figure 3: Architecture distribution.

4.1 Implementation

We implemented CRYPTOREX with 3310 lines of Python code. Apart from that, we integrated the APIs of existing open-source projects to avoid reinventing the wheel. Especially, Binwalk [2] is utilized to unpack firmware images. Angr [1] is used to convert low-level binary code into VEX IR, and Buildroot [3] is used in building cross-file call graphs.

4.2 Experiment Setup

Dataset. During September and October of 2018, we crawled 1327 firmware images in total, covering 12 different IoT vendors, including WD, TP-Link, Linksys, AT&T, Buffalo, and so forth. The device types include IP camera, network attached storage, router, smart plug, smart bulb, and so forth. The detailed data is provided in Table 3.

Among the firmware images we collected, CRYPTOREX successfully unpacked 521 of them (39.3%). Our unpacking implementation relies on Binwalk, which is the de facto tool for firmware unpacking and is able to handle common compression algorithms. However, inevitably, some IoT vendors use non-standard packing techniques such as proprietary compression algorithms and encryption algorithms. In this paper, we did not handle the unsuccessful cases and leave it as future work.

Our further analysis shows that the successfully unpacked firmware samples have seven different architectures – ARM, MIPS, MIPSel, Tilera TILE-GX, PowerPC, MIPS64, and X86_64. It should be diverse enough to demonstrate the capability of cross-architecture analysis of CRYPTOREX. The detailed distribution data is plotted in Figure 3.

Regarding crypto APIs, we extracted all 165 crypto APIs from 7 well-known crypto libraries, as listed in Table 2. In total, we tracked 190 crypto-related arguments.

Table 2: Statistics of used crypto APIs

Library	**# of Crypto APIs**	**# of Tracked Arguments**
cryptlib [4]	3	4
libgcrypt [6]	4	4
libcrypto [42]	80	87
libcrypt [29]	3	4
wolfcrypt [9]	20	30
Nettle [8]	45	46
LibTomCrypt [7]	10	15

Execution environment. Also, in our experiment, CRYPTOREX run on an Ubuntu 16.04 PC equipped with Intel Core i7 quad-core 2.50 GHz CPU and 8G RAM.

4.3 Findings

At the firmware level (Table 3), CRYPTOREX discovered 679 crypto misuse bugs in 126 vulnerable firmware images from 8 vendors, indicating that 24.2% (126/521) of firmware images contain at least one misuse issue. Notably, we found that even 88.7% of Tomato device firmware images are vulnerable. Only the firmware images from Buffalo, Zyxel, TENVIS passed our checking, and no misuse issues were identified.

At the rule level (Table 4), the most common misuse is using ECB mode for encryption (Rule 1, 20.5%), which could result in several security risks. For instance, attackers can determine whether two ECB-encrypted messages are identical. Also, the misuse cases of using constant encryption parameters include constant IV (Rule 2, 4.6%), constant keys (Rule 3, 11.3%), and constant salts (Rule 4, 10.8%). Once they are leaked, the secrecy of stored data and transmitted data could be compromised. Moreover, we found that 4.4% of firmware images use less than 1000 iterations[2] for password-based encryption (Rule 5), indicating that developers tend to sacrifice security to achieve better performance. As a result, attackers can perform brute-force attacks using a large number of candidate passwords.

In our evaluation, we did not find the case of violating Rule 6. Our further investigation shows that the random number generation modules of most IoT firmware images rely on `/dev/urandom` or `/dev/random` directly, without using a random seed. Such implementations are deemed secure.

4.4 Accuracy

False positives. Given the tremendous efforts to manually confirm all 679 identified misuse bugs, we randomly sampled 30 cases from the reported misuses and manually examined them to make inferences about the statistical population. However, although 29 of them (96.7%) were confirmed misusing

[2] In fact, the iteration round of all firmware images violating Rule 5 is 1.

Table 3: Results of crypto misuse detection (by vendors)

Vendor	# of Firmware	# of Unpacked Firmware	% of Unpacked Firmware	# of Vulnerable Firmware	% of Vulnerable Firmware	# of R1 issues	# of R2 issues	# of R3 issues	# of R4 issues	# of R5 issues	# of R6 issues
D-Link	496	201	40.5%	22	11.0%	106	7	30	27	0	0
Linksys	121	70	57.9%	31	44.3%	48	19	41	20	20	0
WD	10	10	100%	4	40.0%	80	36	0	0	3	0
360	5	4	80%	1	25.0%	2	0	0	0	0	0
AT&T	12	0	0%	0	0%	0	0	0	0	0	0
Buffalo	7	4	57.1%	0	0%	0	0	0	0	0	0
Netgear	66	29	43.9%	1	3.5%	11	0	0	0	0	0
TP-Link	47	47	100%	1	2.1%	5	0	0	0	0	0
Tomato	71	71	100%	63	88.7%	102	0	58	58	0	0
MikroTik	23	23	100%	3	13.0%	0	0	6	0	0	0
Zyxel	450	50	11.1%	0	0%	0	0	0	0	0	0
TENVIS	19	12	63.2%	0	0%	0	0	0	0	0	0
Total	1327	521	39.3%	126	24.2%	354	62	135	105	23	0

Table 4: Results of crypto misuse detection (by rules)

Violated Rule	# of Firmware	% of Firmware
Rule 1	107	20.5%
Rule 2	24	4.6%
Rule 3	59	11.3%
Rule 4	56	10.8%
Rule 5	23	4.4%
Rule 6	0	0%
No violation	395	75.8%

crypto functions, there was a false positive for self-defined crypto functions (see Listing 2 extracted from D-Link NAS device DNS-326), which is caused by dead code (i.e., the constant argument is not used in crypto operations). Specifically, in the definition of the self-defined crypto function `sub_155F0()`, if it is invoked with its third parameter being 1 (at Line 19), the low-level crypto function will have its salt parameter being the constant string "1". However, when the self-defined crypto function `sub_155F0()` is invoked at line 5, a "0" is passed as its third parameter. It means that Line 19 is not executed and function `sub_15570()` is being called, and it generates a random string combine with time and PID, the low-level crypto function `crypt()` is not misused.

```
1  signed int __fastcall sub_15270(const char *a1, const char *a2, const char *a3, int a4){
2    int v6,v8;
3    v6 = a4;
4    v8 = (int)getpwnam(a1);
5    if(sub_155F0(v8,v6,0)){
6      ...
7    }
8    ...
9  }
10
11 // self-defined crypto function
12 signed int sub_155F0(int a1, int a2, int a3){
13   ...
14   int v3;
15   char *v5;
16   v3 == a3;
17   ...
18   if(v3 == 1){
19     v5 = "$1$"; //not executed if v3 is 0
20   }else{
21     v5 = (const char *)sub_15570(); //use time() and getpid() to generate salt
22   }
23   v6 = (const char *)sub_14E74(&v11,v5);
24 }
25
26 char* __fastcall sub_14E74(const char *a1, const char *a2){
27   char *v2;
28   v2 = crypt(a1,a2); //low-level crypto function
29   ...
30 }
```

Listing 2: Dead code: D-Link DNS-326.

False negatives. Previous crypto misuse detection approaches [23,26,36,47] focus on mobile platforms, rather than IoT devices. As a result, there is no immediate and labelled dataset as ground truth to quantify the false negative. Despite that, we manually checked the parameters of all crypto API invocations (based on our crypto API dataset as shown in Table 2) in 10 randomly selected firmware images in which no misuse was reported, and we found no false negative.

4.5 Performance

Running CRYPTOREX on 1327 firmware images consumes 7 days and 3 hours (around 171 hours) in total. On average, each

Table 5: Performance analysis

Firmware Model	Firmware Size	# of Analyzed Files	Size of Analyzed Files	Time of Unpacking	Time of CFCG Construction	Time of IR Lifting	Time of ICFG Construction	Time of Taint Analysis	Total Time
E2500	7.0 MB	8	3.12 MB	<1s	3s	10m39s	16s	2m	13m10s
mipsbe-6.42.9	10.7 MB	9	1.11 MB	<1s	3s	1m52s	16s	<1s	2m19s
mipsbe-6.43.2	11 MB	10	1.2 MB	<1s	3s	2m12s	20s	<1s	2m25s
FW_EA6350	16.2 MB	53	3.99 MB	<1s	1m25s	5m18s	1m25s	<1s	9m8s
FW_WRT1900ACv2	32 MB	66	5.4 MB	34s	1m36s	6m41s	2m2s	<1s	11m15s
DSR-250_A2	25.8 MB	64	9.71 MB	3s	1m20s	11m39s	3m34s	4s	19m17s
DCS-960L_A1_FW	9.8 MB	144	6.3 MB	<1s	31s	14m20s	1m52s	<1s	16m19s
My_Cloud_KC2A	103.7 MB	120	26.73 MB	5s	1m57s	28m58s	9m42s	46m19s	98m53s
360P3	8.1 MB	61	4.6 MB	<1s	8s	13m44s	1m45s	<1s	15m43s
360POP-P1	6.7 MB	59	3.7 MB	<1s	8s	26m3s	1m35s	<1s	27m52s
B99_755025	5.9 MB	14	2.8 MB	<1s	4s	10m22s	57s	<1s	11m49s
IPC_V1.7	3.4 MB	1	0.03 MB	<1s	1s	8s	<1s	<1s	12s
NC250_1.0.10	7.6 MB	13	5 MB	2s	1s	16m27s	56s	<1s	17m45s
Archer_C9v1	15 MB	16	4.72 MB	1s	12s	4m58s	1m44s	<1s	7m41s

firmware analysis only costs 1120 seconds (for 521 firmware images that be successfully unpacked).

To understand which factors have significant contributions to the performance, we selected 14 representative firmware samples to investigate, which covers different situations. The firmware selection is based on three factors – firmware size, the number of analyzed files, and the size of analyzed files. In Table 5, we show the information about the analyzed firmware (i.e., the firmware size, the number of extracted files, the number and the size of executables that involve crypto) and the time consumption of each step. It turns out that, in general, IR lifting consumes most of the time. Additionally, we could observe that the larger the size of the analyzed files is, the more time it spends on ICFG construction. Moreover, the time cost of taint analysis is related to the complexity of ICFG directly, especially the number of path branches. For example, since My_Cloud_KC2A contains up to 26.73 MB files, it took 9m 42s to construct ICFG, and further spent 46m 19s on the taint analysis (due to its complex ICFG).

On the other hand, since CRYPTOREX is the first work focusing on the crypto misuse issue in IoT systems, it is inappropriate to make a crosswise performance comparison between our work and the previous ones on mobile platforms (see Section §6.2 for more discussions).

4.6 Case Studies

To further understand the effectiveness of CRYPTOREX and the risk of crypto misuse, we provide three case studies with in-depth analysis.

4.6.1 Tomato Shibby Router

In the firmware of the Tomato Shibby router, CRYPTOREX reported a crypto misuse that is also an authentication bypass vulnerability. We list the vulnerable function in Listing 3. At Line 5, function sub_2493C() invokes nvram_get() to get the encryption key from the NVRAM parameter. However, if such the parameter is not filled, the function will use "admin"[3] as the default encryption key (Line 11, 14, and 19). In the implementation of function crypt, if the second parameter starts with “1”, it will combine the first parameter and the second parameter as the key to encrypt a constant string and output a token. This token will be further used in password checking. This vulnerability could allow attackers to bypass the authentication.

```
1  int sub_2493C()
2  {
3    ...
4    char *dest;
5    v3 = (const char *)nvram_get("
         http_passwd");
6    v4 = v3;
7    strcpy(dest, "$1$");
8    f_read("/dev/urandom", dest + 3, 6);
9    if ( v3 ){
10     if ( !*v3 )
11       v4 = "admin";
12   }
13   else {
14     v4 = "admin";
15   }
16   ...
17   v8 = fopen("/etc/shadow", "w");
18   if ( v8 ) {
19     v9 = crypt(v4, dest); //low-level
           crypto function
20     fprintf(v8, "root:%s:0:0:99999:7:0:0:\
           nnobody:*:0:0:99999:7:0:0:\n", v9)
           ;
21     ...
22     fclose(v8);
```

[3] It is also the default user password.

```
23   }
24   ...
25   sprintf(  (char *)&v11,
26   "root:x:0:0:root:/root:/bin/sh\n"
27   "%s:x:100:100:nas:/dev/null:/dev/null\n"
28   "nobody:x:65534:65534:nobody:/dev/null:/
         dev/null\n", v6);
29   f_write_string("/etc/passwd", &v11, 0,
         420);
30 }
```

Listing 3: Vulnerable code: Tomato Shibby router.

4.6.2 Open-source File Server Netatalk

CRYPTOREX reported crypto misuses on several NAS firmware images. Our further investigation shows that the misuses all occur in the same shared open-source file server called Netatalk. It supports five kinds of authentication ways, which involve random number exchange and Diffie-Hellman key exchange. As shown in Listing 4, for random number exchange, the server uses the user password as a key to encrypt a generated random number and send it to challenge the client. For this case, the ECB mode is used in the DES algorithms (Line 7), which allows attackers to guess random numbers and masquerade as legitimate users easily. In the other function pwd_login() (Line 16) that executes Diffie-Hellman key exchange to negotiate a shared key, we found that a constant IV (Line 20) is used for the CBC encryption (i.e., CAST_cbc_encrypt()). Without using random IV, attackers can easily brute-force the user password.

```
1  /***** uams_randnum.so *****/
2  static int randnum_logincont(void *obj,
       struct passwd **uam_pwd,
3  char *ibuf, size_t ibuflen _U_,
4  char *rbuf _U_, size_t *rbuflen)
5  {
6    ...
7    DES_ecb_encrypt((DES_cblock *) randbuf,
         (DES_cblock *) randbuf,&seskeysched,
          DES_ENCRYPT);
8    ...
9    if (memcmp( randbuf, ibuf, sizeof(
         randbuf) )) { /* != */
10     return AFPERR_NOTAUTH;
11   }
12   ...
13 }
14
15 /***** uams_dhx_passwd.so *****/
16 static int pwd_login(void *obj, char *
       username, int ulen, struct passwd **
       uam_pwd _U_,
17 char *ibuf, size_t ibuflen _U_,
18 char *rbuf, size_t *rbuflen)
19 {
20   unsigned char iv[] = "CJalbert";
21   ...
22   CAST_cbc_encrypt((unsigned char *)rbuf,
         (unsigned char *)rbuf, CRYPTBUFLEN,
         &castkey, iv, CAST_ENCRYPT);
23   ...
24 }
```

Listing 4: Vulnerable code: Netatalk.

4.6.3 OpenSSL

To our surprise, CRYPTOREX also reported a crypto misuse bug in the standard OpenSSL library. As Listing 5 shows, the function EVP_BytesToKey() (Line 22) is used to generate keys and IVs. This function is then invoked with its sixth parameter being 1, which sets the round of iteration to 1. Such an operation violates Rule 5, allowing attackers to perform brute-force attacks to guess the keys and the IVs.

```
1  const OPTIONS enc_options[]=
2  {
3    ...
4    {"k", OPT_K, 's', "Passphrase"},
5    ...
6  };
7    ...
8    int enc_main(int argc, char **argv)
9    {
10     while ((o = opt_next()) != OPT_EOF)
11   {
12     switch (o)
13     {
14       ...
15       case OPT_K:
16       str = opt_arg();
17       break;
18       ...
19     }
20   }
21   ...
22   if (!EVP_BytesToKey(cipher, dgst, sptr,(
         unsigned char *)str, str_len, 1, key
         , iv))
23   {
24     ...
25   }
26   ...
27   if (!EVP_CipherInit_ex(ctx, NULL, NULL,
         key, iv, enc))
28   {
29     ...
30   }
31 }
```

Listing 5: Vulnerable code: OpenSSL.

5 Discussions and Limitations

Though our framework achieves cross-architecture analysis and discovered several cryptographic misuse cases in the wild, there still exist some venues for improvements.

Firmware extraction. In our framework, the firmware will be unpacked to obtain the executables. In the implementation, this step is completed by Binwalk which is the de facto standard for firmware unpacking. However, Binwalk is not a silver bullet for every firmware format: (1) If the firmware images are packed with private compression algorithms which are not covered by Binwalk, the unpacking will fail; (2) Furthermore, if the firmware images are encrypted, Binwalk also cannot unpack them without the correct decryption keys. As a result, due to the non-standard firmware formats, only around 39.3% of firmware samples can be unpacked successfully. This situation limits the scope of our analysis.

Cryptographic function identification. CRYPTOREX uses the API data coming from seven popular cryptographic libraries, which have covered most cases. However, some developers may implement cryptographic algorithms by themselves, i.e., non-standard implementations. The current mainstream cryptographic function identification techniques in binary programs concentrate on dynamic analysis, especially comparing the I/O relationships [14,35]. However, since CRYPTOREX is static analysis solution, these techniques could not be applied directly.

Dynamically generated data. During taint analysis, CRYPTOREX only covers the data stored in binary files. However, if the vulnerable cryptographic parameters are generated dynamically (e.g., received from the Internet or got from NVRAM), CRYPTOREX cannot detect such misuse cases.

IoT apps. Except for firmware in IoT devices, some IoT vendors also provide a mobile app (Android or iOS) to assist the user in controlling the device. In our framework, we do not cover the cryptographic misuse issues in these IoT apps. Some previous work has proposed cryptographic misuse detection solutions for mobile platforms (see Section §6.2).

Automatic repair. As the first step, our technique can identify crypto misuse in IoT devices automatically. However, automated fixing the discovered crypto misuses remains a challenge. One possible solution is to fix the misused function arguments and rewrite the executables automatically. We leave this to future work.

6 Related Work

In this section, we review prior research about security analysis of firmware and misuse of crypto functions.

6.1 Security Analysis of Firmware

In recent years, researchers have a strong interest in the security analysis of firmware in IoT devices [15, 16, 18–21, 25, 27, 39, 45, 48, 49]. On the one hand, prior studies have emphasized addressing the challenges in dynamic analysis and static analysis of firmware. Regarding dynamic analysis, the main challenge lies in the emulation failures caused by diverse architectures and unavailable NVRAMs. To this end, researchers proposed different systems to support firmware emulation [15, 49]. For instance, Chen et al. [15] proposed a dynamic analysis tool which emulates the entire filesystem of Linux-based firmware. Avatar [49] is a framework that orchestrates the execution of an emulator with the real hardware, by forwarding I/O accesses from the emulator to the embedded devices. Apart from that, static firmware analysis also faces the challenge of different architecture [16]. As a result, the solutions to cross-architectural bug search have been widely investigated [25, 27, 45, 48]. For example, Xu et al. [48] proposed a neural network-based graph embedding system that supports multiple firmware architectures and can significantly speed up the vulnerability detection process. However, given that those techniques are designed for discovering general vulnerabilities and focus on scalability, they are less inaccurate for specific vulnerability types. On the other hand, another direction aims to detect specific types of vulnerabilities. For example, Costin et al. [19] performed a large-scale study and found that nearly 10% of the collected firmware images contain bugs in the web interfaces. Shoshitaishvili et al. [46] proposed a model to describe the authentication bypass vulnerability of firmware. David et al. [20] presented an IR-based static analysis solution for finding CVEs in stripped firmware images, which is the most relevant work to our CRYPTOREX. Nevertheless, none of the previous work has tackled the difficulty in identifying crypto misuse of firmware.

6.2 Misuse of Cryptographic Functions

Previous work has noticed the problem of crypto misuse on mobile platforms, say Android and iOS. However, how to detect crypto misuse in IoT systems is an open question.

Egele et al. [23] were the first to perform a large-scale experiment to measure cryptographic misuse in Android apps. Their result showed 88% of tested apps made at least one mistake. Wang et al. [47] analyzed crypto misuse issues in Android native libraries. Muslukhov et al. [40] studied how crypto APIs misuse in Android applications has changed between 2012 and 2016. It provides some updated findings, such as significantly fewer libraries and applications were using ECB mode in 2016. On the other hand, Ma et al. [37] proposed an approach, CDRep, to automatically repair vulnerable Android apps with cryptographic misuses.

Similarly, on the iOS platform, Li et al. [36] designed iCryptoTracer to check cryptographic usage, but the scale of their dataset was quite small. Instead of inspecting a low-level representation of a binary, Feichtner et al. [26] proposed an LLVM-based approach to uncover cryptographic misuse in iOS apps. The result shows that 82% of apps are subject to at least one security misconception.

Also, more recently, Li et al. [34] proposed K-Hunt, a system for identifying insecure keys in x86/64 executables. Different from the above work, CRYPTOREX focuses on the crypto misuse problems in IoT devices. Also, it solves the challenge of cross-architecture crypto misuse checking, not just concentrating on a single platform.

7 Conclusion

In this paper, we introduced the automated cross-architecture analysis framework CRYPTOREX, for detecting cryptographic misuse bugs in IoT devices in a large-scale manner. It utilizes an intermediate representation to solve the issue of non-unified underlying architectures of IoT firmware. Following this trail, in the design of CRYPTOREX, we developed a series of practical techniques to achieve the purpose of reliable crypto misuse detection. Finally, we implemented CRYPTOREX and carried out experiments based on 1327 real-world firmware image samples (from 12 vendors, in 7 different architectures, 521 successfully unpacked ones). CRYPTOREX successfully identified 679 crypto misuse cases, which demonstrates the feasibility of our solution and sheds light on the worrisome situation in today's IoT development.

Acknowledgements

We are grateful to our shepherd Johannes Kinder and the anonymous reviewers for their insightful comments. This work was partially supported by Hong Kong S.A.R. Research Grants Council (RGC) Early Career Scheme / General Research Fund No. 24207815, No. 14217816, National Natural Science Foundation of China (NSFC) under Grant No. 61902148, No. 91546203, Key Science Technology Project of Shandong Province No. 2015GGX101046, Major Scientific and Technological Innovation Projects of Shandong Province, China No. 2018CXGC0708, No. 2017CXGC0704, and Qilu Young Scholar Program of Shandong University.

References

[1] Angr. https://github.com/angr/angr.

[2] Binwalk. https://github.com/ReFirmLabs/binwalk.

[3] Buildroot. https://buildroot.org/.

[4] Cryptlib. https://www.cryptlib.com/.

[5] IDA F.L.I.R.T. Technology: In-Depth. https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml.

[6] Libgcrypt. https://gnupg.org/software/libgcrypt/.

[7] LibTom. https://www.libtom.net/LibTomCrypt/.

[8] Nettle - a low-level cryptographic library. http://www.lysator.liu.se/~nisse/nettle/.

[9] wolfSSL. https://www.wolfssl.com/.

[10] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the Mirai Botnet. In *Proceedings of the 26th USENIX Security Symposium (USENIX-SEC), Vancouver, BC, Canada, August 16-18, 2017*, 2017.

[11] Gogul Balakrishnan and Thomas W. Reps. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction, 13th International Conference, CC 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, 2004.

[12] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011.

[13] Matt Burgess. Google Home's data leak proves the IoT is still deeply flawed. https://www.wired.co.uk/article/google-home-chromecast-location-security-data-privacy-leak, 2018.

[14] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS), Raleigh, NC, USA, October 16-18, 2012*, 2012.

[15] 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), San Diego, California, USA, February 21-24, 2016*, 2016.

[16] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang.

IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 18-21, 2018*, 2018.

[17] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg City, Luxembourg, June 25-28, 2018*, 2018.

[18] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *Proceedings of the 23rd USENIX Security Symposium (USENIX-SEC), San Diego, CA, USA, August 20-22, 2014.*, 2014.

[19] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated Dynamic Firmware Analysis at Scale: A Case Study on Embedded Web Interfaces. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS), Xi'an, China, May 30 - June 3, 2016*, 2016.

[20] Yaniv David, Nimrod Partush, and Eran Yahav. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Williamsburg, VA, USA, March 24-28, 2018*, 2018.

[21] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium (USENIX-SEC), Washington, DC, USA, August 14-16, 2013*, 2013.

[22] Thomas Dullien and Sebastian Porst. REIL: A Platform-independent Intermediate Representation of Disassembled Code for Static Code Analysis. In *The Annual CanSecWest Conference*, 2009.

[23] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS), Berlin, Germany, November 4-8, 2013*, 2013.

[24] Ericsson. Internet of Things forecast. https://www.ericsson.com/en/mobility-report/internet-of-things-forecast, 2018.

[25] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 21-24, 2016*, 2016.

[26] Johannes Feichtner, David Missmann, and Raphael Spreitzer. Automated Binary Analysis on iOS: A Case Study on Cryptographic Misuse in iOS Applications. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec), Stockholm, Sweden, June 18-20, 2018*, 2018.

[27] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), Vienna, Austria, October 24-28, 2016*, 2016.

[28] Josh Fruhlinger. How teen scammers and CCTV cameras almost brought down the internet. https://www.csoonline.com/article/3258748/security/the-mirai-botnet-explained-how-teen-scammers-and-cctv-cameras-almost-brought-down-the-internet.html, 2018.

[29] Free Standards Group. Interfaces for libcrypt. http://refspecs.linuxbase.org/LSB_3.0.0/LSB-PDA/LSB-PDA/libcrypt.html.

[30] Grant Hernandez, Farhaan Fowze, Dave (Jing) Tian, Tuba Yavuz, and Kevin R. B. Butler. FirmUSB: Vetting USB Device Firmware using Domain Informed Symbolic Execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.

[31] Selena Larson. FDA confirms that St. Jude's cardiac devices can be hacked. https://money.cnn.com/2017/01/09/technology/fda-st-jude-cardiac-hack/, 2017.

[32] Chris Lattner and Vikram S. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO), San Jose, CA, USA, 20-24 March 2004*, 2004.

[33] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. Why does cryptographic software fail?: a case study and open problems. In *Proceedings of 5th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys), Beijing, China, June 25-26, 2014*, 2014.

[34] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. K-Hunt: Pinpointing Insecure Cryptographic Keys from Execution Traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS), Toronto, ON, Canada, October 15-19, 2018*, 2018.

[35] Xin Li, Xinyuan Wang, and Wentao Chang. CipherXRay: Exposing Cryptographic Operations and Transient Secrets from Monitored Binary Execution. *IEEE Transactions Dependable Secure Computing*, 11(2):101–114, 2014.

[36] Yong Li, Yuanyuan Zhang, Juanru Li, and Dawu Gu. iCryptoTracer: Dynamic Analysis on Misuse of Cryptography Functions in iOS Applications. In *Network and System Security - 8th International Conference, NSS 2014, Xi'an, China, October 15-17, 2014, Proceedings*, 2014.

[37] Siqi Ma, David Lo, Teng Li, and Robert H. Deng. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS), Xi'an, China, May 30 - June 3, 2016*, 2016.

[38] Roland Moore-Colyer. Samsung SmartThings Hub vulnerabilities leave smart home devices open to attack. https://www.theinquirer.net/inquirer/news/3036719/samsung-smartthings-hub-riddled-with-20-security-bugs, 2018.

[39] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS), San Diego, California, USA, February 18-21, 2018*, 2018.

[40] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. Source Attribution of Cryptographic API Misuse in Android Applications. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (AsiaCCS), Incheon, Republic of Korea, June 04-08, 2018*, 2018.

[41] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI), San Diego, California, USA, June 10-13, 2007*, 2007.

[42] OpenSSLWiki. Libcrypto API. https://wiki.openssl.org/index.php/Libcrypto_API.

[43] Charlie Osborne. Meet Torii, a new IoT botnet far more sophisticated than Mirai variants. https://www.zdnet.com/article/meet-torii-a-new-iot-botnet-far-more-sophisticated-than-mirai/, 2018.

[44] OWASP. Guideline: Testing for Weak Encryption. https://www.owasp.org/index.php/Testing_for_Weak_Encryption_(OTG-CRYPST-004).

[45] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-Architecture Bug Search in Binary Executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, USA, May 17-21, 2015*, pages 709–724, 2015.

[46] 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), San Diego, California, USA, February 8-11, 2015*, 2015.

[47] Qing Wang, Juanru Li, Yuanyuan Zhang, Hui Wang, Yikun Hu, Bodong Li, and Dawu Gu. NativeSpeaker: Identifying Crypto Misuses in Android Native Code Libraries. In *Information Security and Cryptology - 13th International Conference, Inscrypt 2017, Xi'an, China, November 3-5, 2017, Revised Selected Papers*, 2017.

[48] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), Dallas, TX, USA, October 30 - November 03, 2017*, 2017.

[49] 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), San Diego, California, USA, February 23-26, 2014*, 2014.

[50] Nan Zhang, Soteris Demetriou, Xianghang Mi, Wenrui Diao, Kan Yuan, Peiyuan Zong, Feng Qian, XiaoFeng Wang, Kai Chen, Yuan Tian, Carl A. Gunter, Kehuan Zhang, Patrick Tague, and Yue-Hsun Lin. Understanding IoT Security Through the Data Crystal Ball: Where We Are Now and Where We Are Going to Be. *CoRR*, abs/1703.09809, 2017.