Reverse Engineering the Eufy Ecosystem: A Deep Dive into Security Vulnerabilities and Proprietary Protocols

Victor Goeman, Dairo de Ruck, Tom Cordemans, Jorn Lapon, and Vincent Naessens, *DistriNet-KU Leuven*

https://www.usenix.org/conference/woot24/presentation/goeman

This paper is included in the Proceedings of the 18th USENIX WOOT Conference on Offensive Technologies.

August 12–13, 2024 • Philadelphia, PA, USA

ISBN 978-1-939133-43-4

Reverse Engineering the Eufy Ecosystem: A Deep Dive into Security Vulnerabilities and Proprietary Protocols

Victor Goeman
victor.goeman@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Dairo de Ruck
dairo.deruck@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Tom Cordemans
tom.cordemans@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Jorn Lapon
jorn.lapon@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Vincent Naessens
vincent.naessens@kuleuven.be
DistriNet, KU Leuven
3001 Leuven, Belgium

Abstract

The security of Internet-of-Things (IoT) is a growing concern, with IP cameras like those from Eufy promising robust security through military-grade encryption. While Eufy's claims are strong, independent verification of these claims is crucial to confirm the integrity and resilience of its systems against potential vulnerabilities and extend the lessons learned to the broader IoT landscape, ensuring practices keep pace with technological advancements.

We unveiled the inner workings and security measures in the Eufy ecosystem through reverse engineering, particularly focusing on its smart doorbell and Homebase, and evaluated the proprietary peer-to-peer protocol and encryption methods.

This paper offers a comprehensive analysis of the Eufy ecosystem, offering insights into the broader implications of IoT device security. Our investigation revealed critical vulnerabilities within the ecosystem, which were responsibly disclosed and confirmed by Eufy. The vulnerabilities could compromise end-user privacy by allowing unauthorized access to the end users' private network within seconds. A key tool in our research was `dAngr`, a symbolic debugger we developed to facilitate the reconstruction of encryption keys in intricate cross-architecture binaries, thus enabling a more efficient reverse engineering process.

The research revealed vulnerabilities in Eufy's ecosystem, leading to serious privacy and security concerns, and suggests effective countermeasures, stressing the need for continued vigilance in IoT device security.

1 Introduction

Smart doorbells have experienced a remarkable surge in sales, reaching a market value of USD 16.2 Billion in 2023, and are expected to grow at an annual rate of 33.4% from 2023 to 2030 [1]. To give an idea of their popularity, Amazon sold more than 400.000 smart doorbell devices and accessories during the pandemic [34]. Smart doorbells empower end users to monitor and interact with people at their doorstep remotely, enhancing both convenience and physical security. However, this surge in popularity has brought concerns regarding security, safety and privacy to the forefront, forcing doorbell manufacturers to invest in bolstering their security measures.

Eufy [12], a rapidly emerging player, is a part of Anker Innovations. Anker is one of the leading electronics brands in America. Founded in 2016, Eufy is already among the ten most popular IP Camera brands in 2022 [23],

Eufy distinguishes itself with its emphasis on security, offering secure local storage (as it eliminates the need for cloud storage subscriptions), as well as promising military-grade encryption and end-to-end encryption [13].

In this work, we present an in-depth security analysis of the Eufy ecosystem which was studied for more than 9 months. Our research is based on the Eufy Homebase 2 in combination with the Eufy video doorbell 2K. However, our findings extend beyond those devices, affecting a whole array of Eufy devices (including its IP Cameras). Our security analysis includes several techniques including *network analysis, symbolic execution, static and dynamic analysis of the firmware and reverse engineering*. This combined effort of analysis methods enabled us to dissect the complete ecosystem. We exposed a series of critical vulnerabilities across various areas of the ecosystem, encompassing the peer-to-peer protocol, authentication, networking, encryption and the pairing process. These vulnerabilities pose a substantial threat to the ecosystem's confidentiality, integrity and availability.

We present a major attack on the Eufy Homebase, exploiting distinct vulnerabilities, and endangering the network and privacy of all end users using the Eufy ecosystem. The only requirement of the attack is being in proximity (i.e., up to miles away using specialized hardware) of an Eufy device and the attack takes no longer than 20 seconds. No network connection is required. The result of the attack is **unrestricted access** to the end user's home network. Prompt and thorough remediation of these vulnerabilities was of the utmost importance considering the gravity and scale of the attack.

Furthermore, we present `dAngr`, a debugger built upon the symbolic execution engine `angr` [40]. It simplifies and enhances the manual analysis of cross-architecture binaries, abstracting away the complexities of the symbolic execution engine. `dAngr` was used to reconstruct the media AES encryption keys allowing us to recover all video and media sent from the Homebase. Additionally, the ease with which the encryption keys for proprietary peer-to-peer communication can be recovered undermines the assertion of providing military-grade encryption.

To address these vulnerabilities and shortcomings, we propose countermeasures and best practices for each specific issue within the Eufy ecosystem. Following responsible disclosure, Eufy has acknowledged the identified vulnerabilities, and we have provided input to mitigate the various vulnerabilities.

Outline. In the remainder of this paper, we start an overview of the Eufy Ecosystem in Section 2. Section 3 presents the attacker model and methods used for the attacks performed in Section 5. Countermeasures are presented in Section 6 followed by Section 7 with general insights and recommendations. Related work is discussed in Section 8, and we conclude with Section 9.

2 The Eufy Ecosystem

The Eufy ecosystem encompasses several components. First, the smart devices, including the smart- doorbells, lights, cameras, vacuums, entry sensors and more, are connected through a closed and dedicated wireless Eufy network with the Eufy Homebase, a central component of the Eufy ecosystem. This core element of the ecosystem acts as a local hub for the management of connected smart devices and handles encryption, networking, firmware updates, and connects to the cloud through the end users' wired local network. The end user can interact with the Homebase using a mobile App which connects to the Homebase (either through the cloud or via the local network). Alternatively, the Eufy web application can connect to the Homebase through the cloud. The Eufy ecosystem studied in this work is depicted in Figure 1. During our study, we focus on two Eufy devices, namely the Homebase 2 and the video doorbell 2K. However, our findings go beyond these devices, affecting the complete Eufy ecosystem.

Before explaining the details of the various findings and vulnerabilities, we provide essential context for understanding and interpreting the subsequent findings and vulnerabilities related to the Eufy ecosystem.

2.1 Video Streaming and Communication

Commands and video streams are transferred at various points in the ecosystem. Commands consistently use a proprietary peer-to-peer protocol (P2P). Video streams use the P2P protocol or other protocols depending on the network location and application retrieving the stream.

2.1.1 Doorbell Communication

The doorbell solely communicates with the Homebase, and this communication occurs over a dedicated *hidden* Eufy Wi-Fi network. The traffic is secured at the data link layer using wireless communication security. Authentication to this network is established using WPA2-PSK, a pre-shared key of eight characters generated during the initial setup of the Eufy Homebase. This wireless network's SSID follows a pattern consisting of the string *"OCEAN_XXXXXX"* where XXXXXX represents the last 24 bits of the Homebase's MAC address.

Pairing mechanism. The pairing mechanism, as depicted in Figure 2, leverages soundwaves as an out-of-band channel to pass sensitive information to the doorbell. The Eufy App instructs the end user to bring the doorbell in close proximity to the Homebase. Next, the Homebase emits a soundwave carrying both SSID and WPA2-PSK of the dedicated Eufy network. The doorbell retrieves the information contained in the soundwave and connects to the Eufy wireless network. The Homebase and doorbell are subsequently connected via a wireless network protected with a pre-shared key. Afterwards, the Homebase and doorbell exchange their P2P connection information (serial number, licenses, etc.) through the Eufy network, and store this information in flash memory, completing device pairing. Finally, the Homebase and doorbell can exchange commands and pass the camera feed.

Streams and messages. Within this wireless network, various streams and messages are transmitted in the clear. P2P commands allow, for instance, to notify the end user when someone rings the doorbell, or to control the camera of the doorbell. We identified a UDP stream containing a continuous stream of JFIF video data and a smaller TCP stream containing JFIF image data. JFIF can be considered as a successor of the original JPEG format [16].

Once the doorbell and the Homebase are paired, both JFIF streams are persistent, even when the user is not watching the feed. Upon reaching the Homebase, the feed undergoes analysis by a motion detection and facial recognition module. Upon detecting movement, the Homebase promptly notifies the user and subsequently encrypts and stores the video and images locally. Optionally, the user may choose to backup the encrypted media in the cloud.

2.1.2 Communication with the End User

Eufy employs several methods to communicate with the end user depending on the location and application in use. While commands are always sent encrypted over the P2P protocol using a symmetric P2P AES key, the protocols used for transmitting video and images between the Homebase and the end user differ. Figure 3 illustrates the scenarios, which are discussed below in more detail.

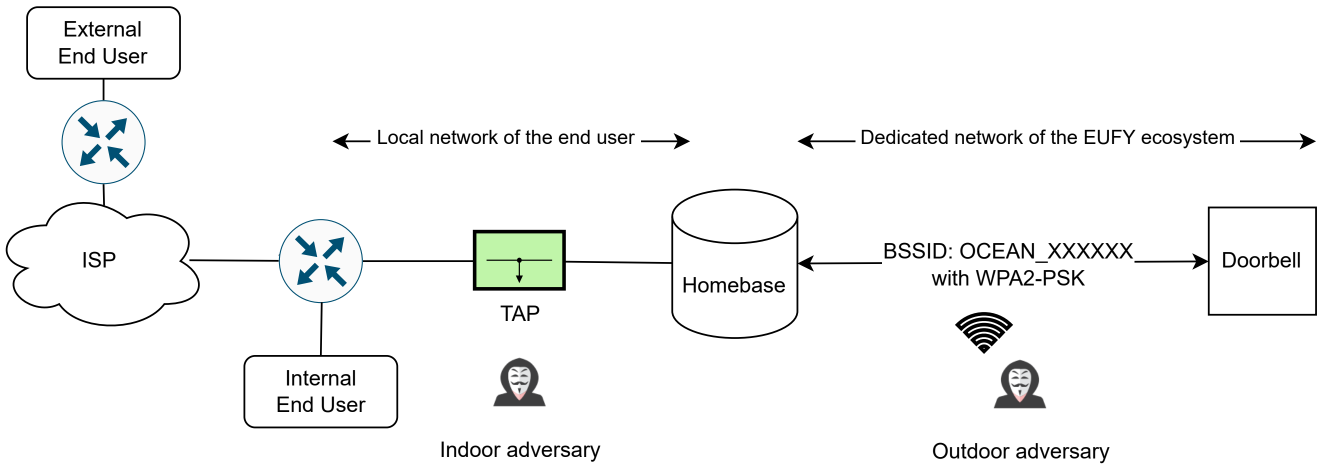
External End User
Local network of the end user
Dedicated network of the EUFY ecosystem
ISP
TAP
Homebase
BSSID: OCEAN_XXXXXX with WPA2-PSK
Doorbell
Internal End User
Indoor adversary
Outdoor adversary

Figure 1: Eufy ecosystem

Mobile App. The first scenario, depicted at the bottom of Figure 3, occurs when the end user uses the mobile App to view the stream. The Homebase employs the proprietary peer-to-peer protocol (P2P) over UDP to communicate the media to the App. The JFIF stream is encrypted with a symmetric P2P AES key. When the end user uses the mobile App to view the stream while connected to the end user's home network (i.e., not using the Internet), communication between both devices is direct. When, on the other hand, the user wants to connect remotely (i.e., a direct connection to the Homebase is infeasible), communication between both devices is relayed through a custom cloud server.

Web viewer. When the end user uses the web viewer (see at the top of Figure 3) independent of the user's location, media is sent using Web Real-Time Communication (WebRTC) [2], an open-source web-based application technology. It is primarily used for establishing real-time, peer-to-peer communication. It is always encrypted using vetted algorithms (e.g., DTLS, SRTP [7, 39]) and leverages standard protocols for NAT Traversal (ICE, STUN and TURN [18, 33, 38]) using another cloud server.

The more secure WebRTC is only used in the communication between the Homebase and the web application. The mobile App always relies on the proprietary P2P protocol to propagate the media stream, and as we will discuss in Section 5.4, these AES keys are insecure, endangering the confidentiality of the ecosystem.

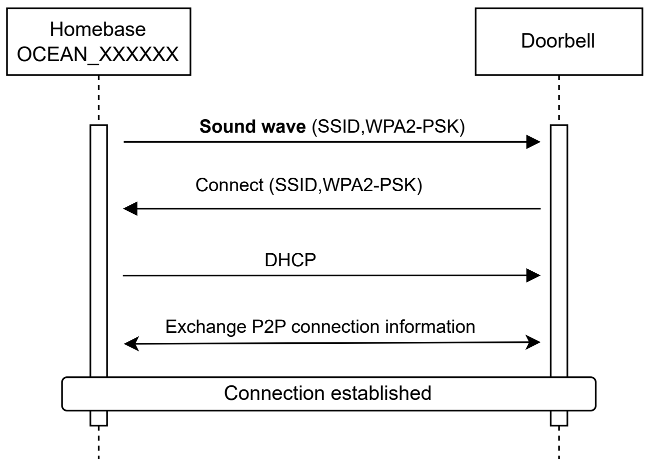

Figure 2: Pairing process

2.2 Homebase Firmware

To understand the functioning of the Homebase, its firmware was thoroughly analyzed. The platform uses a MIPS architecture, running a custom Linux built with Buildroot [3]. Multiple binaries developed by Eufy are present on the firmware. The main binary called `home_security` handles all major functionalities of the Homebase. This binary has multiple instances running concurrently.

3 Attack Vectors & Methods

Our analysis involved various techniques to analyze the Eufy ecosystem. Studying the device from distinct angles allows for a comprehensive analysis of the ecosystem. Our research encompasses three methods of analysis: network analysis, firmware analysis and symbolic analysis. While we conducted an in-depth analysis of the smart doorbell and Homebase, the mobile App and web viewer were only used during network analysis.

Attacker model: In our analysis, we consider two types of adversaries, as illustrated in Figure 1:

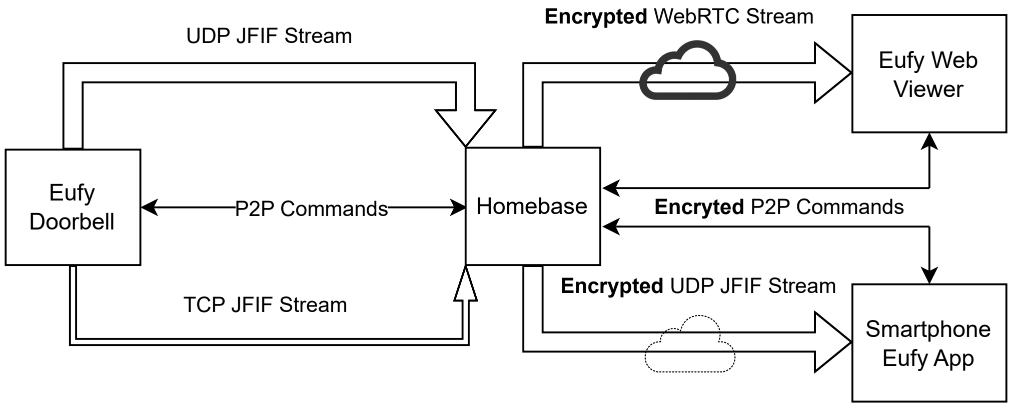

Figure 3: Video streaming in the Eufy ecosystem

1. The *outdoor adversary* operates within reach of the wireless Eufy network but is not connected to either the Eufy or the end user's home network.

2. The *indoor adversary* has access to the home network, including the Homebase. It allows interception and manipulation of communication between the Homebase and other devices in the home network, including the Internet gateway.

Network analysis. The initial phase of our investigation involves the analysis of the network traffic within the Eufy ecosystem. The objective is to uncover the ecosystem's functionality, employed protocols, interactions with external entities, and identification of cleartext communication. This process involves capturing and analyzing the communication between the Homebase and the home gateway along with testing man-in-the-middle attacks (MITM) from an indoor adversary's perspective. From an outdoor adversary's standpoint, the Eufy wireless communication is analyzed using a WiFi dongle.

Firmware analysis - reverse engineering. The process of firmware analysis involves disassembling the devices and scrutinizing debug ports and other hardware peripherals that may facilitate firmware extraction. Upon successful extraction, the firmware undergoes a series of tests, encompassing both automated and manual analysis. Initially, automated analysis of the firmware is conducted using EMBA, an open-source firmware analyzer [24].

Following the automated analysis, an exhaustive manual analysis is performed, selecting proprietary, custom-built binaries for in-depth inspection. To facilitate this reverse engineering step, we employ Ghidra, an open-source reverse engineering tool [11]. Leveraging the Ghidra decompiler to provide insights into the program logic, focusing on areas such as the proprietary peer-to-peer protocol, authentication mechanisms, encryption and decryption processes, and cloud communications.

Selective execution. During our analysis of the key generation mechanism employed for encrypting media, challenges arose during the decompilation of the embedded MIPS binary. Ghidra faced difficulties in generating proper decompiled code for the more intricate functions. While other decompilers may have better support for these types of embedded binaries, we opted for a distinctive approach. We aimed to execute the embedded code to recreate the media keys. However, executing binaries of an embedded device presents its own set of challenges. In the upcoming Section, we introduce the use of `dAngr` to execute a specific function with chosen inputs and retrieve the media encryption key.

4 Selective and Platform Independent Execution with `dAngr`

Running a selected function in an embedded binary allows for several opportunities for testing and analysis. Potential benefits include analysis of the behaviour of the function under various conditions, testing for vulnerabilities, and automated testing procedures. In this Section, we discuss existing techniques' benefits and issues and introduce a novel technique.

4.1 Existing Approaches

Selective execution of embedded functions. The aim of selective execution is to execute a function in a binary of an embedded device, leveraging known inputs to reconstruct, for instance, AES encryption keys. Several techniques exist, each with specific shortcomings. The most challenging approach would be to reconstruct a binary by extracting code using `objdump`. However, this approach is complex and may face difficulties, particularly when dealing with global static variables. A simpler approach involves using a debugger such as radare2 or gdb to execute the required function with the chosen inputs [31]. While with radare2 this may still be complex, requiring the correct memory and register settings, calling a function with chosen arguments in gdb is straightforward.

Unfortunately, a major disadvantage of these solutions is platform dependency. Both binary reconstruction and debuggers require a matching platform and dependencies (e.g., libc) to execute the function.

Platform-dependent execution. Most solutions require either full or partial execution of the binary, necessitating a platform and libraries that match the binary. This requirement poses challenges, particularly for binaries from IoT devices that may use less common platforms. Executing, for instance, a MIPS binary on a standard platform is not straightforward.

To overcome the challenges some solutions propose to run a debugger on the physical device. However, access to such a device may not always be feasible. Another approach involves using a hardware platform with a matching architecture, but this requires a bootable binary that simulates all peripheral initialization and hardware communication, which can be complex and time-consuming. Likewise, emulating the hardware platform using, for instance, QEMU, in which either the binary or the full device may be emulated [29], suffers from this same issue.

An alternative approach is to perform binary lifting into an intermediate representation (IR) and either recompile to another platform or perform virtual execution (interpretation) of the IR code. However, both approaches suffer from issues with simulating actual devices.

Our approach. To address the shortcomings of the existing solutions, we combine both *Selective execution* and *Platform-agnostic execution* using dAngr, a debugger for angr (see Section 4.2). Platform independence is achieved through the execution of VEX IR. Furthermore, to overcome the configuration and hardware initialization challenges, we use selective execution to simulate only the necessary functionality to execute the selected functions.

Table 1 summarizes the main points of comparison between the traditional methods for selective and platform-independent execution of embedded functions versus our approach using dAngr. The dAngr approach combines the advantages of both supporting selective execution and being platform agnostic, offering a more streamlined and versatile solution for testing and analyzing embedded devices.

4.2 dAngr: a Debugger for angr

To support our research, we developed dAngr, a debugger built on top of angr. angr is a symbolic execution engine implemented in Python. angr handles the complexities of binary lifting and interpretation, while the debugger interfaces (a command line, and JSON interface) simplify its use, requiring minimal knowledge about the underlying engine.

While for our attack we use dAngr for concrete execution (i.e., with concrete inputs instead of symbolic inputs), it also supports symbolic execution. The debugger contains common debugging commands such as adding, removing, enabling/disabling breakpoints, stepping and running. Note that since we use angr as an interpreter, stepping occurs per basic block instead of per instruction as in other debuggers. Similar to other debuggers, the run command performs execution until the next breakpoint or the end of the binary. However, in the case of symbolic execution, the debugger may stop if a forking state is reached, allowing the user to choose the branch to take.

Moreover, dAngr supports setting and retrieving registers or memory at specified addresses. Instead of starting the execution upon the binary's entry point, it is possible to relay the start of the execution to a selected address. Combining memory and registry control with starting execution at a chosen address enables selective execution of a function in the binary.

However, this approach still requires some in-depth knowledge of the platform e.g., to specify the function arguments using the correct registers and calling convention. To simplify executing functions and make our tool more accessible, we support three additional commands: set_function_prototype, to specify the function prototype, set_function_call to set the debugger to the function address and correctly set the arguments, and *get_return_value* to retrieve the return value with the type as specified in the prototype. Listing 1 shows an example of the commands to execute a function with specified arguments.

Listing 1: dAngr example commands for calling a function func with arguments

```
> set_function_prototype int func(char*, int)
> set_function_call func({"abc",2)
> run
> get_return_value
```

In addition, testers can pass hooks to the debugger to replace or implement specific functions called inside the executed code. This feature was used to debug the binary and attest certain parameters.

This novel approach simplifies the execution process and aligns seamlessly with our goal of recovering the key generation algorithm, allowing the execution of a specific function with concrete inputs without the need for an exhaustive and complex setup.

dAngr is made open-source and available on GitHub [1].

[1]https://github.com/angr-debugging/dAngr

Table 1: Comparison of Execution Approaches

Attribute/Approach	**Existing Techniques**	`dAngr` **Approach**
Selective Execution	Requires binary reconstruction or debugger (e.g., radare2, gdb)	Enables execution of specific functions with chosen inputs
Platform Independence	Highly dependent, requiring matching platform and libraries	Utilizes VEX IR for platform independence
Complexity of Setup	Complex, may involve binary reconstruction or setting memory and registers	Simplifies simulation, avoiding deep knowledge of platform specifics
Simulation of Hardware/Peripheral Setup	Requires accurate simulation or emulation for execution	Only simulates the necessary functionality for executing selected functions
Suitability for Testing and Analysis	Limited by platform dependency and setup complexity	Enhanced by ease of executing specific functions and platform independence
Approach to Execution	Direct execution on hardware or through emulation/simulation	Virtual execution of intermediate representation (IR) code

5 Attacking and Identifying Vulnerabilities in the Ecosystem

After analyzing the Eufy doorbell and Homebase, we uncovered several flaws that undermine the security of the system. In the following, we introduce the steps taken during the analysis of the Eufy Homebase and smart doorbell.

5.1 Firmware Acquisition

To gain access to the Eufy ecosystem, our first step involves acquiring the firmware.

Homebase. For the Homebase, the firmware acquisition process commences with the disassembly of the device, revealing UART debug ports. Connecting a USB-to-TTL reader to these ports, we discovered a password-protected UART shell. However, during the boot process, a temporary recovery shell can be accessed without authentication. Within this recovery shell, we analyzed the filesystem, unveiling the password of the root user. This password coincided with the WPA2-PSK securing the dedicated Eufy network, highlighting a vulnerability in *password reuse*. These particular flaws had been previously identified by other researchers conducting security analysis on the Eufy ecosystem [5]. By using this password, we gain entry to the password-protected UART shell, thereby accessing the system. Consequently, the firmware is obtained through a firmware dump in this shell.

Doorbell. Unlike the Homebase, the doorbell lacks obvious UART ports. However, upon carefully analyzing the hardware of the doorbell, SPI NOR flash is detected. Reading this NOR flash chip is accomplished using an SPI reader. Specifically, we used a CH314a flash programmer to acquire the firmware of the doorbell [22].

While the doorbell primarily handles only essential functions, such as capturing and sending media to the Homebase, the Homebase manages more intricate operations such as media processing, encryption and authentication. Hence, the majority of the security analysis is concentrated on the Homebase. Notably, an automated analysis of the firmware using EMBA did not uncover any significant vulnerabilities.

5.2 Cracking the WPA2-PSK

To understand the construction of the WPA2-PSK key, we located, through reverse engineering, the responsible functions in the `home_security` binary found in the Homebase firmware.

The WPA2-PSK of the Homebase has a fixed length of eight characters, comprising both lowercase letters, uppercase letters, plus, slash and numbers (64 possible values), yielding a theoretical entropy of approximately 48 bits. This is already a weak key strength to protect wireless network communication. Nevertheless, as shown in Table 2, brute-forcing a key with *only commodity hardware* may take several years.

However, weaknesses in the key generation process lower the entropy of the WPA2-PSK even further. The *WPA2-PSK has a one-to-one mapping* with a non-secret variable (i.e., the serial number), making it susceptible to exploitation. Learning or brute-forcing the serial number compromises the security of the dedicated Eufy network.

The serial number can be discovered in at least three distinct ways: it is printed on the device casing, it can be intercepted in the LAN or WAN network when using the App, and it can be brute forced.

The WPA2-PSK is derived by (1) computing the MD5 hash

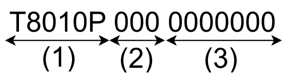

Figure 4: Serial number format

of this serial number, (2) encoding the result using Base64, and (3) taking the first eight bytes.

$$\text{WPA2-PSK} = \text{B64}(\text{MD5}(serial))[0:7]$$

An example of a key generation is shown below:

$$\begin{aligned}
\text{SN} &= T8010P23224107B0 \\
\text{MD5(SN)} &= c4732772b06902fe671689ef92946675 \\
\text{B64(MD5(SN))} &= YzQ3MzI3NzJiMDY5MDJmZTY\ldots \\
\text{WPA2-PSK} &= \text{B64(MD5(SN))}[0:7] \\
&= YzQ3MzI3
\end{aligned}$$

To brute force the WPA2-PSK by guessing serial numbers, we can take advantage of the structure of the serial number. Upon scrutinizing the serial numbers of over 30 distinct Homebases found online, a pattern emerges. Figure 4 shows the various parts of the serial number: (1) device-type identifiers, (2) batch identifiers and (3) device-specific identifiers.

The device-type identifiers are uniform across distinct Homebases. The only difference we detect is the character following the device-type identifiers (i.e., `T8010`), which may be either "P" or "N". For the batch identifiers, we consistently observe numbers ranging from zero to three. The device-specific identifiers consist of seven hex numbers. However, extrapolating the entire range based on only 30 analyzed devices may lead to inaccuracies. Therefore, we adopt a pragmatic approach, considering both a best-case (BC) scenario, where only batch identifiers are considered to be in the range of zero to three, and a worst-case (WC) scenario, where batch identifiers consist of all possible hexadecimal values. While the former, using our commodity hardware takes approximately 11 hours, the latter may need 30 days.

Upon further investigation of the WPA2-PSK key generation process, we discovered an even worse vulnerability allowing the recovery of the WPA2-PSK in only 20 seconds.

A closer look at the code revealed that only the first six characters of the MD5 hash affect the WPA2-PSK: by definition of Base64 encoding, the first eight characters of the Base64 encoded string only depend on the first six characters of the MD5 hash. To make things worse, the MD5 hash function outputs hexadecimal characters. Hence, the WPA2-PSK is based on only six hexadecimal characters (16 possible values each). The example above simplifies to the following:

	Possibilities	Entropy (Bits)	Time to crack*
WPA2PSK	$64^8 = 2.18 * 10^{15}$	48	17 years
Serial number (WC)	$2 * 16^3 * 16^7 = 2.2 * 10^{12}$	41	30 days
Serial number (BC)	$2 * 4^3 * 16^7 = 34 * 10^9$	35	11 hours
Shortened MD5	$16^6 = 16.8 * 10^6$	24	20 seconds

Table 2: Entropy of distinct WPA2-PSK phases
* (Using our commodity hardware testing @ ± 800K keys/sec)

$$\begin{aligned}
\text{SN} &= T8010P23224107B0 \\
\text{MD5(SN)} &= c4732772b06902fe671689ef\ldots \\
\text{MD5(SN)}[0:5] &= c47327 \\
\text{B64(MD5(SN)}[0:5]) &= YzQ3MzI3 \\
&= \text{WPA2-PSK}
\end{aligned}$$

This vulnerable key derivation process diminishes the WPA2-PSK's entropy to a mere 24 bits. This low entropy allows the creation of a custom password list containing all ±16.8 million potential eight-character passwords for Eufy's dedicated networks. By exploiting this vulnerability, an attacker can leverage a dictionary attack on the WPA2-PSK.

Exploit: Executing a brute force attack on a dedicated Eufy network protected with a WPA2-PSK involves cracking the four-way handshake between the client and access point (i.e., the Homebase). To obtain this handshake, a de-authentication attack on the doorbell is executed [17]. Simultaneously de-authenticating the doorbell and monitoring the wireless network enables us to capture the handshake. Such attacks are common for leveraging an offline attack on WPA and WPA2 security protocols. While brute-forcing WPA3 is more challenging, the key's limited entropy of only 24 bits makes a brute-force attack feasible, even in the case of WPA3.

The Aircrack-ng tool suite is employed for this purpose [26]. Subsequently, the captured handshake is cracked offline using Hashcat [30], a password-cracking tool, and our custom password list. Using commodity GPU hardware [2], Hashcat successfully cracks the WPA handshake within 20 seconds.
The vulnerability in the WPA2-PSK generation has been assigned CVE-2023-37822.

5.3 Lack of Network Security

Insecure communication. Armed with the password list crafted earlier, the previously deemed secure dedicated Eufy network now faces a significant threat. Engaging in wardriving for hidden wireless networks with an SSID starting with *"OCEAN_"*, we gain unrestricted access to the network of

[2]We reach about 800k tests per second using the AMD Ryzen 9 5950X, NVIDIA RTX 3080 10GB, 64GB DDR4 3600Mhz

any Eufy Homebase 2 ecosystem. Furthermore, all communication, including commands, video streams and images, is sent unprotected, in *cleartext* on this dedicated network. Compromising the confidentiality and integrity of the Eufy ecosystem.

Lack of isolation. Despite the insecure communication, the most critical networking flaw of the Homebase lies in its use as a pivot. The *lack of isolation* between the dedicated network and the end user's home network through the Homebase is a significant vulnerability. Acting as a proxy, the Homebase permits traffic to flow from the dedicated Eufy network to the end user's home network without any restrictions. Since the end user lacks visibility into the dedicated Eufy network, a malicious actor joining this network could go unnoticed. In such a scenario, the complete private network of the user becomes accessible to the outdoor attacker, turning the Eufy ecosystem into an easy and stealthy entry point for adversaries into the private networks of end users. When combined with the vulnerabilities discussed earlier, the potential consequences of this flaw become enormous.

5.4 Breaking the Encryption Schemes

Eufy states to ensure the users' privacy and promises military-grade encryption. This Section delves into the subsequent measures taken by Eufy to achieve these goals and how we compromised them.

AES encryption in Eufy's ecosystem. Eufy predominantly uses the weaker ECB version of the AES encryption as its cryptographic foundation for data protection. Within the ecosystem, two distinct symmetric AES encryption keys play pivotal roles in safeguarding user information: the P2P AES encryption key dedicated to *securing P2P communication* (commands and messages), and the media key used to *encrypt media* (images and videos for both storage and communication). Both the Homebase and client applications reconstruct the AES keys using obscure key generation processes.

Before discussing the distinct key generation methods, it is imperative to introduce the so-called PPCS identifier (also denoted as *PPCS_ID*). Next to the serial number, this device-specific identifier is stored in flash memory. It consists of three distinct parts separated by dashes, among others, used to derive encryption keys. The 20-character string contains three parts: the first part consists of uppercase characters that identify the device type, the second part is made up of unique numbers related to the device, and the third part contains uppercase characters. It takes the following format:

$$\texttt{AAAAAAA} - \texttt{000000} - \texttt{BBBBB}$$

5.4.1 Breaking Encrypted P2P Traffic

Encrypted P2P traffic exchanged between the Homebase and the mobile App (local or remote) uses AES ECB encryption. The AES key is created containing device-specific information as follows:

$$Key = PPCS\_ID[0:15] + serial[9:15]$$

Here, the key is a combination of the first 16 characters of the PPCS identifier, and the last seven characters of the serial number. It is crucial to note that all parameters used in the key derivation process can be observed in the network traffic between the Homebase and the App (local or remote). This information is transmitted in plain text. Given that all key material is pre-shared over the same network, the encryption of the P2P traffic brings no additional security.

5.4.2 Generating the Media Encryption Key

In the encryption of media, Eufy adopts a more intricate method for generating the AES encryption keys. Although the encryption of videos and images slightly differ (i.e., storage format), the encryption key is the same. This Section focuses on the encryption process of images to demonstrate how the Eufy ecosystem secures its media.

Encrypted image format. Before delving into the encryption process of an image, it is essential to understand the format of an encrypted image. An encrypted image is a file that starts with a plaintext Eufy header containing the serial number of the Homebase (*serial*) and a random value (*rand*). Both are used in recovering the encryption key. The format of the Eufy header is represented as follows:

$$eufysecurity :< serial >: 01 < rand >:$$

The cleartext Eufy header is succeeded by 256 encrypted bytes, being the encrypted JFIF header. Subsequently, this encrypted header is followed by the remainder of the unencrypted JFIF image. Since only the JFIF header is encrypted, leaving the rest of the image unencrypted, there is a potential for information leakage.

Media AES encryption key generation algorithm. The reconstruction of the media encryption key is more complex. Since the Homebase binaries do not contain code to decrypt media, we focus on the encryption process. The primary function responsible for encrypting images is denoted `jpg_encrypt`. This function first constructs the encryption key using `create_pic_code_v1`, and next, encrypts the image.

The generation of the media key entails three steps, depicted in pseudocode in Algorithm 1. First, a Homebase unique *baseCode* is created based on the serial number and

Algorithm 1 Critical key generation functions in the `create_pic_code_v1` algorithm (pseudo code)

```
1:  function GETHOMEBASECODE($serial, PPCS\_ID$)
2:      $sfx = \texttt{getPPCSSuffix}(PPCS\_ID)$        ▷ See Alg. 2
3:      $baseCode = \texttt{concat}(serial[0:l], \texttt{str}(sfx))$
4:      return $baseCode$
5:  end function
6:  function GETRANDSEED($PPCS\_ID$)
7:      $sfx = \texttt{getPPCSSuffix}(PPCS\_ID)$
8:      $rndStr = "01"||\texttt{str}(\texttt{random}())||\texttt{str}(1000 - sfx)$
9:      $seed = \texttt{Obfuscate}_1(\texttt{MD5}(rndStr))$
10:     return $(seed, rand)$
11: end function
12: function CREATEIMAGEKEY($baseCode, seed$)
13:     $h = \texttt{SHA256}("01" + baseCode + seed)$
14:     $encKey = \texttt{Obfuscate}_2(h)$
15:     return $encKey$
16: end function
```

PPCS identifier. Next, a *seed* is generated from the same PPCS identifier along with a freshly generated random integer. Finally, the encryption key is derived from the combination of *baseCode* and the *seed*.

Both `genHomebaseCode` and `genRandSeed` use *PPCS_ID* to compute a suffix sfx. Then, the *baseCode* is constructed by concatenating a substring of the serial (where the length l depends on the last byte) with this suffix. Similarly, the *seed* is computed by concatenating the random integer with a value derived from the suffix (i.e., $1000 - sfx$). The resultant string is then hashed, followed by an obfuscation step. This obfuscation is essentially transforming bytes. Finally, in `createImageKey`, the *baseCode* and the *seed* are hashed and again obfuscated with a custom deterministic algorithm. It is evident that each of these steps, including the obfuscations, is reproducible given the *serial*, *PPCS_ID* and *rand* are known.

5.5 Reconstructing the Media Encryption Key Using `dAngr`

As outlined earlier, the intricacies involved in the encryption key derivation pose significant challenges to manual reverse engineering. The first attempts were ineffective due to the complexity and inaccuracies found in the decompiled Ghidra code. Specifically, the Ghidra decompiler encountered difficulties with certain sections of the MIPS code, resulting in unreliable decompiled output.

As discussed in Section 4.1, we adopt a novel approach to recover the encryption keys. We leverage `dAngr` for a concrete and platform agnostic execution of the `create_pic_code_v1` function required to reconstruct the key. Listing 2 shows the commands passed to `dAngr` to reconstruct encryption keys given the correct inputs.

Since the binary only contains the encryption part, we use this function to reconstruct the keys. However, in this case, during the actual key generation, a fresh random value is generated to create a unique key for each image. To be able to decrypt an encrypted image, we need to reconstruct the key given a specific random (included in the Eufy header in the encrypted image). Therefore, we take advantage of the hooking functionality of `dAngr` to replace the call to generate a random (i.e., `random`) with a stub that returns the random number included in the encrypted image.

Listing 2: `dAngr` commands to reconstruct a media encryption key

```
> load_hooks hooks.py
> set_function_prototype void
    create_pic_code_v1(char*, int, char*,
    char*, char *)
> set_function_call create_pic_code_v1('
    T8010P123DEADBEA', 0x10, 'ZYXABCD
    -456789-TSRQP', '0'*10, '0'*32)
> run
> get_string_memory 0x5000
```

In Listing 2, a function is set up and configured with the correct parameters. The first argument is the serial number, the third is the *PPCS_ID* and the final two parameters are character arrays of 10, respectively 32 characters for the returned random value and the encryption key. The first 16 characters of the latter hold the actual key. To read out the key, we need to access the memory of the last parameter of which the address ($0x5000$) is printed during debugging.

To decrypt a given image, we extract the *serial* and *rand*, and together with the *PPCS_ID*, we can easily recover the encryption key. The PPCS identifier can be intercepted from the network traffic between the Homebase and the mobile App.

Alternatively, we can eliminate the dependency on the *PPCS_ID*. After further investigation, we successfully recovered the algorithm to generate the sfx suffix derived from the PPCS identifier (see Algorithm 2).

Algorithm 2 `getPPCSSuffix`(*PPCS_ID*)

```
1: $s = PPCS\_ID[0:15].split('-')[1]$
2: $sfx = \texttt{int}(s[0]) + \texttt{int}(s[1]) + \texttt{int}(s[3]) + \texttt{int}(s[5])$
3: if $sfx < 5$ then
4:     $sfx = sfx * 2$
5: end if
6: return $sfx$
```

This function calculates the sum of four of the six digits in the middle part of the PPCS identifier. Next, the result is doubled when the sum is smaller than five, further diminishing the already limited entropy. Thus, instead of monitoring the network traffic and waiting for the *PPCS_ID* to leak, we can opt for a brute-force approach on the parts of the *PPCS_ID* being used.

We simply hook the `getPPCSSuffix` function and generate the 480 potential sfx values output by `getPPCSSuffix`. We can easily verify the correctness of a key based on the presence of the magic bytes (i.e., 0xFFD8 for JFIF images) in the decrypted JFIF header. Once we find a match, we also have a valid sfx which can be used to decrypt further images.

Using this brute-force approach, we can decrypt any encrypted image without requiring any additional information beyond the encrypted image itself.

The media key derivation process is clearly flawed, enabling an indoor attacker or the cloud server to decrypt all media. It is important to note that other researchers independently uncovered the encryption mechanism while reverse engineering the mobile App [6]. Their motivation primarily focused on facilitating access to the ecosystem through open-source tools. In contrast, our objective was to identify weaknesses in their encryption process. Notably, we achieved this goal, even generating Eufy keys leveraging the lack of entropy without requiring the device identifiers.

6 Countermeasures

Considering the vulnerabilities outlined in the previous section, defining countermeasures for fortifying the Eufy ecosystem is crucial. For each vulnerability, we propose countermeasures:

- *Password reuse*: Avoid reusing the WPA2-PSK. To protect the UART boot sequence, the debug port should be disabled.

- *Password has one-to-one mapping*: Ensure passwords do not have a one-to-one mapping with *public* variables. These variables should be kept secret. Alternatively, passwords should be randomly chosen using a secure random generator.

- *Low entropy password*: Although rectifying low-entropy passwords presents a challenge. A solution would be to discreetly transmit a new high-entropy WPA2-PSK to each paired device after updating each Eufy device. This must be done before changing the Wireless network, allowing background updates without user interaction or breaking the connection with the smart devices.

- *Lack of isolation*: Prevent attackers from pivoting between networks by implementing Linux `iptables` functionality on the Homebase. The Homebase should act solely as an Internet gateway restricting traffic to flow between isolated networks. If necessary, only essential ports should be forwarded.

- *Cleartext traffic*: Augment WPA2-PSK as a protection mechanism with additional protection. Encrypt network communication using established protocols such as TLS to introduce an extra layer of security and end-to-end encryption. This should be implemented for all communication, including P2P traffic.

- *Bad encryption keys*: Enhance the key derivation process, by adopting standard and secure key derivation and encryption schemes. Refrain from using proprietary DIY algorithms and AES ECB mode. Furthermore, a proper key management solution must be implemented such that keys must not be derived from non-secret information such as serial numbers.

  Additionally, instead of only encrypting the media headers, the entire payload should be encrypted.

7 General Insights & Recommendations

Conducting an in-depth security analysis has provided valuable insights into the Eufy ecosystem, unveiling both its vulnerabilities and strengths. Several key lessons can be learned from this comprehensive examination.

To evaluate the impact of our research, we initially assessed the alignment of the Eufy Homebase and doorbell with the OWASP IoT Top 10 [28] before and after our investigation. Initially, the Eufy ecosystem showcased strong compliance with the OWASP IoT Top 10, boasting standardized AES encryption and secure WPA2-PSK-protected network communication. Only an unprotected UART recovery shell and having open UART debug ports were left unaddressed. However, as revealed in our analysis, critical flaws in data encryption, network architecture weaknesses and the use of weak guessable passwords are uncovered. Consequently, Eufy's standing on the Top 10 shifted after our in-depth analysis, now failing in several key areas.

Our analysis revealed that in IoT, particularly in the realm of consumer IoT, security is still often treated as an afterthought, especially in teams lacking security expertise. This often results in reliance on security by obscurity and do-it-yourself (DIY) solutions.

For instance, our work revealed several weak key generation methods. Notably seeking compliance with security standards such as EN 303 645 (ETSI Consumer IoT) may not have been sufficient to prevent the flaws discovered in this study. The recommendation stemming from these insights is clear: IoT manufacturers should invest in comprehensive IoT security training. They must adopt industry-standard, vetted protocols to comply with established security standards. Rather than resorting to custom solutions, strict adherence to best practices is crucial.

Enforcing unique keys per device has become mandatory for compliance with prominent security standards. In turn, security compliance will be required to enter markets worldwide. For instance, embedded devices can only be sold on the EU market after having received a CE label, and demonstrating security compliance will be part of the certifying process from August 2024. The aforementioned requirement – i.e. unique device keys – is imposed by the realistic attacker model in which a malicious stakeholder with physical access to one IoT device cannot undermine the whole ecosystem. Well-established mechanisms and protocols exist and many standards point to very concrete tactics (without enforcing a specific solution or technology).

However, many developers still develop proprietary solutions instead of relying on widely recognised mechanisms. The major reason is the often recurring complex tension between security and manageability. To decrease the key management burden, obscure mechanisms are often constructed in which keys are unique per device but can still be derived by having knowledge of the device firmware. This implies that an attacker with physical access to an IoT device no longer directly undermines the security of the whole ecosystem (as keys are no longer shared across devices) but can indirectly derive the keys of other devices by inspecting the device firmware. This is possible if an attacker has physical access to one device and can rely on firmware inspection tools which are becoming easily accessible. To tackle this evolution, standards and even legislation should become stricter in the sense that they do not only impose requirements concerning general characteristics of the device but also on feasible and non-feasible mechanisms to enforce it. Although this may restrict the degrees of freedom at design and development time and may result in more advanced key management (ultimately resulting in a more expensive lifecycle), it will result in improved security.

The community, particularly in consumer IoT, would benefit from the availability of *reference architectures and proof of concepts* that depict commonly encountered use cases and scenarios. These should encompass essential aspects such as the proper use of STUN, TURN and ICE services for remote access; secure pairing of smart devices, gateways and mobile devices; correct use and implementation of public key infrastructure; secure update procedures; and the secure use of cloud services and API's. Such resources would deter developers from resorting to DIY strategies and obscure solutions.

Securing IoT devices is undoubtedly a substantial endeavor, requiring expertise across various domains, including embedded hardware and software, network security, cloud communication, and mobile or web development. The commitment to strong security practices is essential for the sustained integrity of IoT ecosystems.

8 Related Work

We discuss in this Section relevant and previous research and studies that influenced our approach to analyzing the Eufy ecosystem.

State of the art of IoT security. Costin et al. performed the first large-scale analysis on IoT devices [10], examining over 683 firmware images, unveiling vulnerabilities on 123 distinct products. Another large-scale analysis is done by Neshenko et al. [25], they focus on discovered IoT vulnerabilities and classify the various vulnerabilities and weaknesses inherent to IoT devices. Performing new large-scale analyses has become increasingly challenging, due to a recent trend where manufacturers strive to maintain the secrecy of their device's firmware. This approach may result in fostering security through obscurity, which fails to deter attackers equipped with sufficient resources.

In a more targeted study, Schwartz et al. analyze the security of 16 popular IoT devices leveraging reverse engineering techniques [41]. Their systematic application of reverse engineering techniques uncovered distinct vulnerabilities, emphasizing the importance of this method in identifying security weaknesses. Obermaier et al. focus on cloud-based video surveillance systems, analyzing four distinct IP Cameras [27] through a combination of network and firmware analysis. This approach led to the uncovering of various vulnerabilities related to authentication, proprietary encryption algorithms and weak certificate validation. Rondon et al. delved into the security of E-IoT systems scrutinizing proprietary protocols used in E-IoT settings [36]. Collectively, these studies indicate the urgent need for enhanced security measures in IoT devices.

WPA attacks. The landscape of wireless security, particularly in the context of Wi-Fi networks, has been a subject of extensive research and exploration. Lorente et al. scrutinized WPA2 password-generation algorithms in Wireless routers and discovered that many routers used weak password-generation algorithms [21]. Reversing the algorithms, Lorente et al. discovered that in most algorithms known parameters were used as input and that they had a simple deterministic password-generation process. One of the vulnerabilities we uncovered is similar to this work. However, we go further than discovering a one-to-one mapping, identifying multiple weaknesses in Eufy's password generation algorithm.

Reversing engineering. Several studies discuss methodologies for discovering and analyzing vulnerabilities [14, 19, 20, 42]. In this related work, reverse engineering is considered an efficient but exhaustive method for analyzing embedded devices. Techniques for more efficient reverse engineering and methodologies for performing a complete device analysis are discussed. Thomas et al. present a framework to reduce the upfront effort in analyzing and reverse engineering using static and dynamic analysis techniques [45].

In case studies, Casagrande et al. applied reverse engineering methodologies to unveil vulnerabilities in the Xiaomi ecosystem. Their work exposed issues in both the pairing process and in applications developed by Xiaomi [8, 9]. In their work, they reverse-engineered the Xiaomi companion App and the Bluetooth Low-Energy communication. The reverse engineering led to the uncovering of various vulnerabilities in the pairing process of the Xiaomi Fitness tracking system and the Xiaomi E-scooters. The vulnerabilities they discovered are cleartext keys, unauthenticated pairing and modifying the password without authentication. Ullrich et al. reversed the Neato vacuum and discovered an attack leveraging weak secret keys and a buffer overflow via the cloud to break the Neato ecosystem [46]. Giese et al. reverse engineer using hardware hacking the Amazon Echo Dot and perform IoT forensics to uncover bad practices that lead to personal data leakage [15]. Other examples of high-impact attacks uncovered by reverse engineering are the Zigbee worm exploiting Philips Hue lamps [37], and breaking glucose monitoring systems thanks to weak proprietary protocols [35]. In our research, we employ a similar methodology to uncover vulnerabilities, focusing on reversing the binaries, networking, and internal operations of the Eufy ecosystem. Contrary to the above work, we present a novel approach leveraging a new cross-platform debugger to assist manual reverse engineering in embedded devices.

Symbolic execution. Yadegari et al. and Banescu et al. emphasize symbolic execution as a potent mechanism to circumvent obfuscation techniques [4, 48]. Symbolic execution proves invaluable in identifying weaknesses and vulnerabilities. Nevertheless, symbolic execution faces its own set of challenges, notably in the analysis of cryptographic functions, which is inherently complex. Vanhoef et al. demonstrate that simulating cryptographic primitives during symbolic execution can be done to find weaknesses in cryptographic functions [47]. Ramos et al. develop an under-constrained symbolic execution framework to analyze individual functions rather than whole programs, bypassing several weaknesses of symbolic execution engines [32]. Contrary to prior work, this work leverages the binary lifting and interpretation provided by `angr` to make debugging platform-independent. While our attack only requires concrete execution, our debugger also supports symbolic execution.

Case studies including the Eufy doorbell. P. Moore analyzed the web interface of the Eufy doorbell [44]. Moore proved that Eufy uploads images to the cloud without authorization. Moore also discovered that the video stream of the Eufy doorbell was sent unencrypted. These vulnerabilities were confirmed and patched by Eufy, ensuring that now all video and images are end-to-end encrypted.

M.A. Stanislav examined various security frameworks to determine the overall security posture of internet-connected devices [43]. An analysis was performed on 40 internet-connected cameras. This analysis includes information gathering, disassembling the device, analyzing the various interfaces of the device, and more. Eufy is one of the cameras being analyzed, and comes out as one of the more mature brands, having overall good security and conforming to best practices.

The open-source community also reverse-engineered the Eufy App and reconstructed the P2P protocol. Allowing them to replace the App or web interface, and connect it to a home automation system [6]. The project primarily focuses on a specific aspect of the P2P protocol related to communication between App and Homebase. However, the P2P protocol within the ecosystem extends further than App Homebase communication, the communication between the Homebase and other Eufy devices is a critical part of the P2P protocol. To build upon and expand the existing research of the Eufy ecosystem, we dissect the firmware of the Eufy devices and the Eufy ecosystem internals, while actively seeking vulnerabilities and weaknesses.

9 Conclusion

The reverse engineering and analysis of the Eufy ecosystem provided insights into the intricate workings of its devices. This investigation uncovered multiple weaknesses, highlighting critical areas of one of the top players in the IP Camera domain. The core of our work involved the analysis of the proprietary peer-to-peer protocol, dissecting the encryption mechanisms, and understanding the internal network's behaviour through a combination of reverse engineering, binary interpretation, and network traffic analysis.

We introduce a novel approach for key reconstruction in embedded devices. We developed `dAngr` a symbolic debugger that augments manual reverse engineering. By leveraging `angr`, a symbolic execution engine that implements binary lifting and interpretation of the lifted code, our tool enables platform-agnostic execution of specific functions, allowing us to execute isolated cryptography functions in an embedded cross-architecture binary without a complex or time-consuming process. We demonstrate our novel approach by reconstructing the AES keys for media encryption in the Eufy ecosystem.

Our findings culminated in an attack on the Eufy ecosystem requiring no network connectivity. The sole prerequisite is proximity to the Homebase's dedicated network. Leveraging two vulnerabilities uncovered during our analysis, the attack serves as a potential entry point into the end user's private home network. The ease and severity of this attack deem it highly critical. We proposed appropriate countermeasures for each identified flaw.
Eufy has confirmed the vulnerabilities and initiated security patches, further bolstering their security.

Responsible disclosure. In June 2023, we responsibly disclosed all newly identified vulnerabilities to Eufy. The comprehensive disclosure process was conducted through Anker's channels. In addition to providing a detailed write-up of the vulnerabilities, we included recommendations for effectively mitigating these issues. A condensed version of the recommended mitigations can be found in Section 6.

References

[1] MarkNtel Advisors. Smart Doorbell Market Achieves USD 16.2 Billion 2023, Braces for 16.7% CAGR Elevate Until 2030. https://www.marknteladvisors.com/research-library/smart-doorbell-market.html, 2024.

[2] Harald T. Alvestrand. Transports for WebRTC. RFC 8835. https://www.rfc-editor.org/info/rfc8835, January 2021.

[3] Buildroot Association. Buildroot. https://buildroot.org/docs.html, 2024.

[4] Sebastian Banescu, Christian Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 189–200. ACM.

[5] Gerhard Hechenberger Bernhard Gründling and Steffen Robertz. Sec consult - the eufycam long-term observation. https://sec-consult.com/blog/detail/the-eufycam-long-term-observation/, 2024.

[6] Patrick Broetto. eufy-security-client. https://github.com/bropat/eufy-security-client, 2024.

[7] Elisabetta Carrara, Karl Norrman, David McGrew, Mats Naslund, and Mark Baugher. The Secure Real-time Transport Protocol (SRTP). RFC 3711. https://www.rfc-editor.org/info/rfc3711, March 2004.

[8] Marco Casagrande, Riccardo Cestaro, Eleonora Losiouk, Mauro Conti, and Daniele Antonioli. E-spoofer: Attacking and defending xiaomi electric scooter ecosystem. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 85–95. ACM, 2023.

[9] Marco Casagrande, Eleonora Losiouk, Mauro Conti, Mathias Payer, and Daniele Antonioli. Breakmi: Reversing, exploiting and fixing xiaomi fitness tracking ecosystem. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(3):330–366, Jun. 2022.

[10] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large-scale analysis of the security of embedded firmwares. SEC'14, page 95–110, USA, 2014. USENIX Association.

[11] NSA's Research Directorate. Ghidra. https://ghidra-sre.org/, 2024.

[12] Eufy. Eufy. https://us.eufy.com/, 2024.

[13] Eufy. Privacy Commitment. https://us.eufy.com/pages/privacy-commitment, 2024.

[14] Aurélien Francillon, Sam L. Thomas, and Andrei Costin. Finding software bugs in embedded devices. In Gildas Avoine and Julio Hernandez-Castro, editors, *Security of Ubiquitous Computing Systems: Selected Topics*, pages 183–197. Springer International Publishing.

[15] Dennis Giese and Guevara Noubir. Amazon echo dot or the reverberating secrets of IoT devices. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 13–24. ACM.

[16] Eric Hamilton. JPEG File Interchange Format Version 1.02. https://www.w3.org/Graphics/JPEG/jfif3.pdf, 1992.

[17] Stefan Savage John Bellardo. 802.11 denial-of-service attacks:real vulnerabilities and practical solutions. page 95–110, USA, 2003. USENIX Association.

[18] Ari Keränen, Christer Holmberg, and Jonathan Rosenberg. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. RFC 8445. https://www.rfc-editor.org/info/rfc8445, July 2018.

[19] Xixing Li, Qiang Wei, Zehui Wu, and Wei Guo. A comprehensive survey of vulnerability detection method towards linux-based IoT devices. In *Proceedings of the 2023 2nd International Conference on Networks, Communications and Information Technology*, pages 35–41. ACM.

[20] Muqing Liu, Yuanyuan Zhang, Juanru Li, Junliang Shu, and Dawu Gu. Security analysis of vendor customized code in firmware of embedded device. In Robert Deng, Jian Weng, Kui Ren, and Vinod Yegneswaran, editors, *Security and Privacy in Communication Networks*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 722–739. Springer International Publishing.

[21] Eduardo Novella Lorente, Carlo Meijer, and Roel Verdult. Scrutinizing WPA2 password generating algorithms in wireless routers. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association.

[22] mantech. EEPROM CH341A 24 25 Series Flash BIOS USB Programmer. https://www.mantech.co.za/Datasheets/Products/CH341B-Programer.pdf, 2024.

[23] MarkWideResearch. Smart Doorbell Market Analysis. https://markwideresearch.com/smart-doorbell-market/, 2024.

[24] Michael Messner. EMBA. https://github.com/e-m-b-a/emba, 2022.

[25] Nataliia Neshenko, Elias Bou-Harb, Jorge Crichigno, Georges Kaddoum, and Nasir Ghani. Demystifying iot security: An exhaustive survey on iot vulnerabilities and a first empirical look on internet-scale iot exploitations. *IEEE Communications Surveys and Tutorials*, 21(3):2702–2733, 2019.

[26] The Aircrack ng Project. Aircrack-ng. https://www.aircrack-ng.org/, 2024.

[27] Johannes Obermaier and Martin Hutle. Analyzing the security and privacy of cloud-based video surveillance systems. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security*, IoTPTS '16, page 22–28, New York, NY, USA, 2016. Association for Computing Machinery.

[28] OWASP. Owasp iot top 10. https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Top_10, 2024.

[29] Qemu project. QEMU. https://www.qemu.org/, 2024.

[30] The Hahscat Project. Hashcat. https://hashcat.net/hashcat/, 2023.

[31] radare org. Radare2. https://rada.re/n/index.html, 2024.

[32] David A. Ramos and Dawson Engler. Under-Constrained symbolic execution: Correctness checking for real code. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 49–64, Washington, D.C., August 2015. USENIX Association.

[33] Tirumaleswar Reddy.K, Alan Johnston, Philip Matthews, and Jonathan Rosenberg. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 8656. https://www.rfc-editor.org/info/rfc8656, February 2020.

[34] Grand View Research. Smart Doorbell Market Size, Share and Trends Analysis Report By Product Type (Wired Doorbell, Wireless Doorbell), By End-user (Residential, Commercial), By Region, And Segment Forecasts, 2023 - 2030. https://www.grandviewresearch.com/industry-analysis/smart-doorbell-market-report, 2024.

[35] Luca Reverberi and David Oswald. Breaking (and fixing) a widely used continuous glucose monitoring system. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[36] Luis Puche Rondon, Leonardo Babun, Ahmet Aris, Kemal Akkaya, and A. Selcuk Uluagac. LightningStrike: (in)secure practices of e-IoT systems in the wild. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 106–116. ACM.

[37] Eyal Ronen, Colin O'Flynn, Adi Shamir, and Achi-Or Weingarten. IoT goes nuclear: Creating a ZigBee chain reaction. *IEEE Security and Privacy*, pages 54–62.

[38] Jonathan Rosenberg, Christian Huitema, Rohan Mahy, and Joel Weinberger. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489. https://www.rfc-editor.org/info/rfc3489, March 2003.

[39] Henning Schulzrinne, Anup Rao, Rob Lanphier, Magnus Westerlund, and Martin Stiemerling. Real-Time Streaming Protocol Version 2.0. RFC 7826. https://www.rfc-editor.org/info/rfc7826, December 2016.

[40] 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 *IEEE Symposium on Security and Privacy*, 2016.

[41] Omer Shwartz, Yael Mathov, Michael Bohadana, Yuval Elovici, and Yossi Oren. Opening pandora's box: Effective techniques for reverse engineering IoT devices. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications*, volume 10728, pages 1–21. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[42] Omer Shwartz, Yael Mathov, Michael Bohadana, Yuval Elovici, and Yossi Oren. Reverse engineering IoT devices: Effective techniques and methods. 5(6):4965–4976. Conference Name: IEEE Internet of Things Journal.

[43] Mark A. Stanislav. *Multi-dimensional Security Integrity Analysis Of Broad Market Internet-connected Cameras*. PhD thesis, Dakota State University, 2022.

[44] Nicholas Sutrich. Security researcher says Eufy has a big security problem. https://www.androidcentral.com/accessories/smart-home/security-researcher-says-eufy-has-a-big-security-problem, 2022.

[45] Sam L. Thomas, Jan Van den Herrewegen, Georgios Vasilakis, Zitai Chen, Mihai Ordean, and Flavio D. Garcia. Cutting through the complexity of reverse engineering embedded devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021:360–389, Jul. 2021.

[46] Fabian Ullrich, Jiska Classen, Johannes Eger, and Matthias Hollick. Vacuums in the cloud: Analyzing security in a hardened IoT ecosystem. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*, Santa Clara, CA, August 2019. USENIX Association.

[47] Mathy Vanhoef and Frank Piessens. Symbolic execution of security protocol implementations: Handling cryptographic primitives. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, Baltimore, MD, August 2018. USENIX Association.

[48] Babak Yadegari and Saumya Debray. Symbolic execution of obfuscated code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 732–744. ACM.