Camveil: Unveiling Security Camera Vulnerabilities through Multi-Protocol Coordinated Fuzzing

Fuchen Ma[*], Yuqiao Yang[†], Yuanliang Chen[*], Yanyang Zhao[*], Ting Chen[†✉], Yu Jiang[*✉]
[*]*School of Software, Tsinghua University, KLISS, BNRist, Beijing, China*
[†] *University of Electronic Science and Technology of China, Chengdu, China*

***Abstract*—Security cameras are widely deployed in safety-critical environments, supporting real-time video streaming and device control via protocols such as RTSP, ONVIF, and HTTP. Vulnerabilities in these systems can lead to frozen video feeds or surveillance failures, potentially resulting in property or safety losses. While fuzzing is a useful technique for discovering vulnerabilities, existing protocol and IoT fuzzers typically treat each protocol independently, overlooking the cross-protocol dependencies present in real-world cameras.**

To address this gap, we propose `CAMVEIL`, a fuzzing framework designed to uncover vulnerabilities in security cameras through multi-protocol coordinated fuzzing. The key insight is that certain protocols can modify the internal state of the camera, indirectly affecting the behavior of other protocols, making some vulnerabilities only discoverable through state-dependent, cross-protocol interaction. To exercise such interactions, `CAMVEIL` builds a protocol-aware camera status model that abstracts internal camera states and defines their dependencies across protocols. Guided by this model, `CAMVEIL` generates coordinated test sequences to explore interleaved protocol behaviors. Additionally, it integrates a logic-aware monitoring component that continuously analyzes response packets to detect semantic inconsistencies or abnormal control flows. Using this approach, `CAMVEIL` has discovered 22 previously unknown vulnerabilities across 9 industrial camera models from Hikvision, Honeywell, TP-Link, FOSCAM, EZVIZ, and Santachi. These flaws could allow attackers to disrupt live video streams or disable camera functionality, potentially causing critical surveillance failures.

1. Introduction

Security cameras play a vital role in safety-critical environments such as industrial facilities, energy infrastructure, and hospitals, where real-time video monitoring is essential for operational safety and incident response. These devices communicate with clients via standard network protocols, including ONVIF [1], HTTP [2], and RTSP [3], to support functionalities such as live video streaming, camera control, and configuration. Through these protocols, clients can retrieve video feeds, adjust viewing angles, and issue device-specific commands.

[✉]*Yu Jiang and Ting Chen are the corresponding authors.*

In real-world deployments, camera protocols often interact by accessing and modifying shared system states. These interactions arise when multiple clients simultaneously issue commands through different protocols that manipulate common internal resources. For example, as illustrated in Figure 1, an HTTP client may send a `continuous_move` request that changes the camera's movement status from idle to moving and updates its position values. At the same time, an ONVIF client may issue a `move_up` command that also modifies the movement and position states. Similarly, an RTSP client may initiate video streaming via a `PLAY` request, while an ONVIF client concurrently resets the encoding method, both operations affecting the encoding state of the camera.

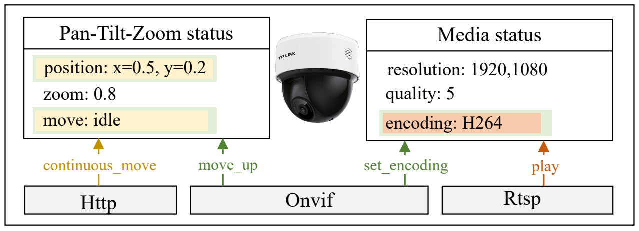


Figure 1. Camera protocols always interact with each other. HTTP and ONVIF may access the position and move state of system 'Pan-Tilt-Zoom' status. While, ONVIF and RTSP may achieve the encoding value in the media status.

These cross-protocol interactions reflect a tightly coupled runtime environment, where changes made by one protocol can influence or conflict with the behavior of others. Such inter-dependencies are a potential source of logic bugs and race conditions. A real-world example is CVE-2023-3959 [4], a critical vulnerability discovered in Zavio security cameras. This bug is triggered when concurrent ONVIF and RTSP requests access shared resources, such as the encoding status, leading to XML parsing errors and real-time video feed interruptions. With a CVSS score of 9.8, the vulnerability highlights the risks introduced by uncoordinated protocol behavior. This also underscores the importance of systematically exercising and testing cross-protocol interactions to uncover such vulnerabilities.

Fuzzing has become a mainstream technique for vulnerability discovery in IoT devices and protocol implementations. IoT device fuzzers such as IoTFuzzer [5] and DIANE [6] rely on companion apps to generate syntactically valid inputs that reach deep code paths. Protocol fuzzers like Peach [7] and Bleem [8] focus on specific protocol structures to construct test cases. However, both lines of work typically treat protocols in isolation, lacking the coordination needed to trigger state-dependent bugs that arise from protocol interactions. To effectively uncover vulnerabilities in security cameras, two key challenges must be addressed.

The first challenge lies in effectively generating test messages for multiple protocols in a way that can exercise their interactions. A straightforward approach might involve running multiple fuzzers in parallel, each targeting a different protocol such as RTSP, ONVIF, or HTTP. However, in the absence of a coordination mechanism or scheduling logic, the messages generated by these independent fuzzers lack inter-dependencies and fail to induce the protocol interactions that occur in real-world deployments. As a result, these independent messages do not effectively trigger complex protocol interactions within the camera, leading to low testing effectiveness.

The second challenge is detecting non-crashing bugs when they occur. Since camera firmware is often inaccessible, failures may not be immediately apparent, even if a test message triggers errors within the device. Tools like IoTFuzzer and DIANE address this issue by monitoring heartbeat responses or TCP disconnections, but these tools can only detect vulnerabilities that cause camera crashes. Other logical bugs, such as video feed freezing, go undetected by existing methods. Therefore, timely detection of these bugs is a significant challenge in camera testing.

To address these challenges, we propose CAMVEIL, a fuzzing framework designed to uncover vulnerabilities in security cameras through multi-protocol coordinated testing. To tackle the first challenge, CAMVEIL constructs a *Camera Status Model* that captures key runtime states of the camera, including PTZ (pan-tilt-zoom) [9] status, media configuration, and system parameters. This model defines the interactions and dependencies between these states, and CAMVEIL labels messages from protocols such as ONVIF, HTTP, and RTSP with their corresponding state transitions. By doing so, it generates semantically linked test sequences across protocols, effectively exercising complex cross-protocol behaviors and improving fuzzing depth. To address the second challenge, CAMVEIL integrates a *Logic Vulnerability Monitor*. This monitor continuously probes the camera using standard operations and observes behavioral anomalies such as frozen video streams, or inconsistent state updates. By semantically checking runtime status changes, CAMVEIL is able to detect subtle logic bugs that would otherwise evade traditional crash-based detection methods.

We implemented CAMVEIL and evaluated it on 9 popular industrial cameras from TP-Link, Hikvision, FOSCAM, Honeywell, EZVIZ, and Santachi. So far, CAMVEIL has identified 22 previously unknown vulnerabilities. These vulnerabilities could have serious consequences. For example, one issue identified in the TP-Link security camera model TL-IPC433H-A4-W10 can cause the video feed to freeze, allowing attackers to disrupt any RTSP clients connected to the camera with well-crafted packets. To compare, we also tested these devices using DIANE and Peach, running separate fuzzing instances on different protocols, but they detected no vulnerabilities. When we equipped Peach with the monitoring mechanisms designed for CAMVEIL, it identified only 3 bugs. These results demonstrate the effectiveness of CAMVEIL's multi-protocol coordinated fuzzing approach.

Our paper makes the following contributions:

- We propose a method for multi-protocol coordinated fuzzing to detect vulnerabilities in security cameras.
- We design and implement CAMVEIL[1]. By leveraging a camera status model and a semantic monitor, CAMVEIL generates interrelated test messages and effectively detects logical bugs in cameras.
- We evaluate CAMVEIL on 9 industrial security cameras from 6 vendors. To date, CAMVEIL has detected 22 previously unknown vulnerabilities.

2. Protocols in Security Cameras

Security cameras typically use network protocols to communicate with clients. While some vendors implement proprietary protocols alongside specific applications [10], most security cameras rely on public protocols to ensure compatibility with NVR (Network Video Recording) devices and VMS (Video Management Systems) [11]. Common public protocols used by security cameras include ONVIF [1], RTSP [3], and HTTP [2].

ONVIF is a global standard that promotes interoperability among IP-based security devices, enabling seamless communication between cameras, recorders, and other equipment from different manufacturers. As of 2024, over 30,000 products comply with the ONVIF protocol [12], illustrating its widespread adoption in the security camera industry. RTSP is a network protocol designed to control the transmission of video and audio streams. Typically, RTSP operates in conjunction with RTCP [13] and RTP [14]: while RTSP manages the streaming session, RTP handles the actual transport of media data. RTCP, on the other hand, provides monitoring and feedback on the quality of the transmission. In security cameras, HTTP typically functions as a web server, enabling users to access camera settings and live feeds through a web browser.

All protocols in a security camera interact by concurrently modifying or accessing the system status. For example, ONVIF can perform many of the same functions as the HTTP web server, such as rotating the camera or adjusting various configurations. This overlap allows multiple protocols to manage similar settings, which can lead to conflicts. Both ONVIF and RTSP, for instance, can control media transmission by sending commands like 'play' or 'stop', potentially resulting in overlapping commands if used simultaneously. Additionally, RTSP and HTTP both access and modify the camera's media status; if a user changes the encoding settings via the HTTP web interface,

1. Camveil at: https://anonymous.4open.science/r/CamVeil-15FB

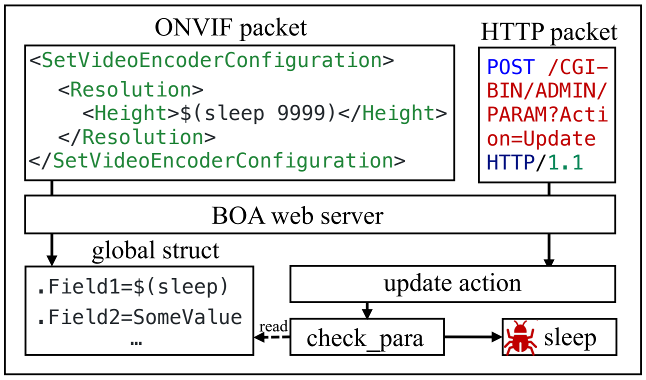
the RTSP clients need to respond to these changes to ensure seamless functionality. Such interactions among protocols require careful management to avoid conflicts that could disrupt the camera's operation.

3. Overview

3.1. A Motivating Example

In this section, we gave a motivating example to show how multi-protocol coordinating bugs are triggered and why existing work fail to detect them. The case is based on the bug CVE-2023-3959 [4], which has a CVSS score of 9.8. It is a logic bug found in Zavio security cameras.

Vulnerability Triggering. As shown in a prior public analysis [15], CVE-2023-3959 stems from protocol interactions between ONVIF and HTTP in Zavio cameras, where conflicting operations cause XML parsing errors. Figure 2 illustrates the detailed steps involved in triggering this vulnerability. The process begins with a crafted ONVIF packet that embeds attacker-controlled input, such as the command `sleep 9999`, which stalls the internal update logic and may lead to video stream freezing. This packet tries to set the video encoder configuration of the camera. It is processed by the ONVIF handler in the Zavio IP camera firmware. During handling, the malicious payload is stored into the first field of a global structure.



```
<SetVideoEncoderConfiguration>
  <Resolution>
    <Height>$(sleep 9999)</Height>
  </Resolution>
</SetVideoEncoderConfiguration>
```

```
POST /CGI-
BIN/ADMIN/
PARAM?Acti
on=Update
HTTP/1.1
```

```
.Field1=$(sleep)
.Field2=SomeValue
…
```

Figure 2. A motivating bug case where an ONVIF packet and a HTTP packet trigger a command injection bug together.

Concurrently, an HTTP request is issued to trigger the update action on the encoder too, which invokes the `check_para` function. Within this function, the code accesses the previously populated global structure and passes the unvalidated user input to the `popen()` system call for execution. As a result, this sequence leads to command injection: arbitrary shell commands embedded in the ONVIF configuration can be executed via the HTTP interface, ultimately allowing unauthenticated remote code execution.

Challenges to Detect this Bug. Detecting the vulnerability illustrated in Figure 2 presents two major challenges: *Challenge 1*: First, triggering the bug requires coordinated interaction between multiple protocols: ONVIF and HTTP. A naive multi-protocol fuzzing approach might launch independent fuzzers for each protocol, such as RTSP, ONVIF, or HTTP, in parallel. However, without a scheduling mechanism that enforces causal or temporal dependencies across protocols, these fuzzers fail to generate sequences that reflect realistic inter-protocol workflows. Detecting the behaviors in this case necessitates a fuzzing strategy that is aware of cross-protocol state propagation and interaction semantics. Existing fuzzing tools do not model such interdependencies, leading to poor coverage of interaction-induced bugs.

Challenge 2: Second, even if the malicious sequence is successfully triggered, identifying the bug is non-trivial due to the absence of immediate crash symptoms. Unlike memory corruption vulnerabilities that result in segmentation faults or device crashes, this bug may silently disrupt the device's behavior. Conventional detection methods, such as monitoring TCP disconnections, heartbeat failures, or service crashes, are insufficient to capture such non-crashing logic flaws. Consequently, detecting these issues requires semantic-level observation of device functionality, such as responsiveness of the video feed, which remains a largely unaddressed challenge in embedded system fuzzing.

Requirements to Detect such Bugs. Based on the two challenges discussed earlier, we identify two essential requirements for a fuzzing framework to effectively detect such vulnerabilities. First, the framework must support coordinated fuzzing across multiple protocols. This requires awareness of how different protocol messages affect shared device state, and the ability to schedule messages accordingly. Second, the framework must be capable of detecting non-crashing logic bugs. The system should incorporate semantic-level monitoring techniques, such as checking for video liveness, execution side effects, or inconsistent configuration responses, to uncover these subtle failures.

3.2. Threat Model

We consider an attacker attempting to exploit security vulnerabilities in IP cameras through remote access. The threat model is defined as follows:

Attacker Capabilities. The attacker does not need to fully compromise the underlying operating system or obtain root access. Instead, the focus is on abusing exposed application-layer behaviors through valid protocol interfaces. The attacker has network access to the target device, either via the same local network or through an exposed public interface (e.g., ONVIF or HTTP services accessible over the Internet). The attacker is capable of crafting and sending arbitrary messages conforming to standard protocols supported by the camera, such as ONVIF, HTTP, and RTSP.

Attack Vectors. Given the ability to send crafted messages over multiple protocols, attackers can exploit inconsistencies and interactions across protocol boundaries to trigger unintended behavior. For example, a malicious ONVIF message can be used to modify the device's internal state, which is later consumed by an HTTP request that processes it without proper validation. Similarly, a sequence of legitimate-looking RTSP and ONVIF messages may lead

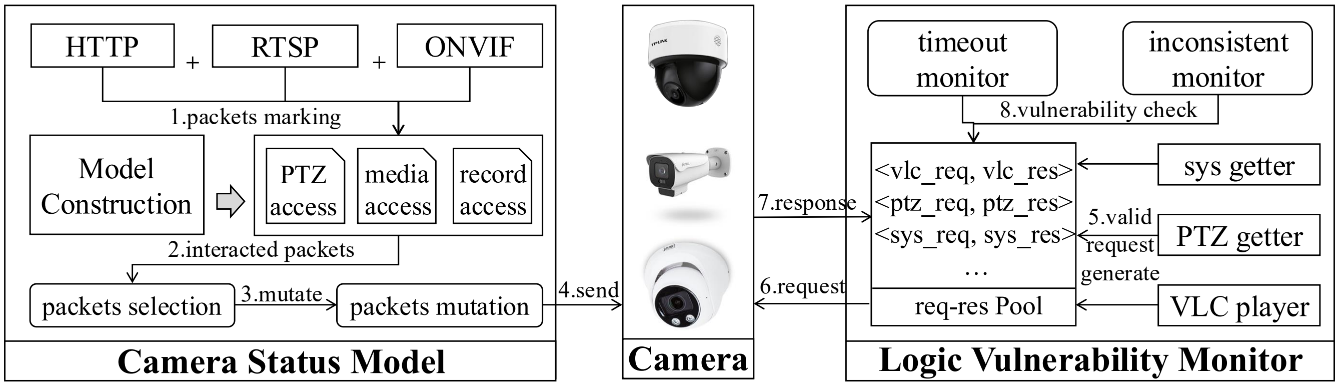


Figure 3. The overall workflow of CAMVEIL. Based on the camera status model, CAMVEIL labels protocol packets with the specific statuses they attempt to access, then selects and mutates interacting packets as test inputs. Additionally, CAMVEIL employs a logic vulnerability monitor to detect vulnerabilities by checking if valid packets time out and verifying that responses remain consistent with previous ones.

the device into a specific internal state that causes logic flaws, such as service freezing, unauthorized command execution, or persistent misconfiguration. These attacks rely not on protocol-level exploits in isolation, but on the subtle interplay between independently functioning modules.

4. Design of Camveil

Design Goal: A practical security camera vulnerability detection framework should have the following properties:

- **Cross-Protocol Interaction Coverage.** The framework should enable effective exploration of interactions across multiple protocols. Many real-world vulnerabilities in IP cameras arise from the interplay between different protocols. The system must be able to generate semantically meaningful and state-aware message sequences that reflect realistic multi-protocol workflows and expose interaction-induced bugs.
- **Detection of Non-Crashing Logic Bugs.** The framework should be capable of identifying functional vulnerabilities that do not lead to system crashes. These include command injection, configuration inconsistencies, and video stream freezing. To detect such subtle issues, the system must incorporate semantic monitors that analyze runtime behavior and detect anomalies beyond simple connection loss or exceptions.
- **Black-Box Compatibility.** Since many commercial IP cameras provide limited access and closed-source firmware, the framework must work under black-box settings, where collecting code coverage is infeasible. It should rely only on externally observable behaviors and publicly exposed protocol interfaces, without requiring instrumentation, debug symbols, or root access. This ensures practical applicability to off-the-shelf devices.

4.1. Overall workflow

Figure 3 presents the overall workflow of CAMVEIL, which consists of two main components: the *Camera Status Model* and the *Logic Vulnerability Monitor*. The workflow proceeds as follows: 1) For each supported protocol, HTTP, RTSP (including RTP and RTCP), and ONVIF,CAMVEIL collects valid packets from real-world clients and labels them according to the internal camera status they access (e.g., PTZ, media, system). 2) In each fuzzing round, CAMVEIL selects a subset of packets that are likely to trigger cross-protocol interactions based on the camera status model. 3) The selected packets are then mutated while preserving protocol semantics, using knowledge of the protocol-specific data structures. 4) The mutated packets are sent to the target camera. 5) In parallel, a VLC player, a PTZ status retriever, and a system status retriever generate standard requests to query the camera's runtime status. 6) These status-checking requests are sent to the camera. 7) When responses are received, they are stored in a request-response pool, indexed for later analysis. 8) Two logic monitors analyze the collected responses: a timeout monitor detects unresponsive behavior, while an inconsistency monitor identifies semantic anomalies (e.g., frozen video, incorrect PTZ state). These checks enable detection of non-crashing logic vulnerabilities. We will describe each component in detail in the following sections.

4.2. Camera Status Model

4.2.1. Camera Status Model Construction. To define the resources and global runtime state of a security camera, we first construct a comprehensive camera status model. This model categorizes the device's internal status into seven distinct sub-statuses, each representing a key functional dimension of camera behavior. Figure 4 illustrates the structure of this model and the dependencies between different statuses. We now describe each sub-status in detail:

- **PTZ Status**: Represents the pan-tilt-zoom state of the camera. It includes four fields: *position*, which indicates the current camera angle; *zoom*, which denotes the magnification level of the lens; *move*, which reflects whether the camera is actively rotating; and *speed*, which captures the rotation velocity.

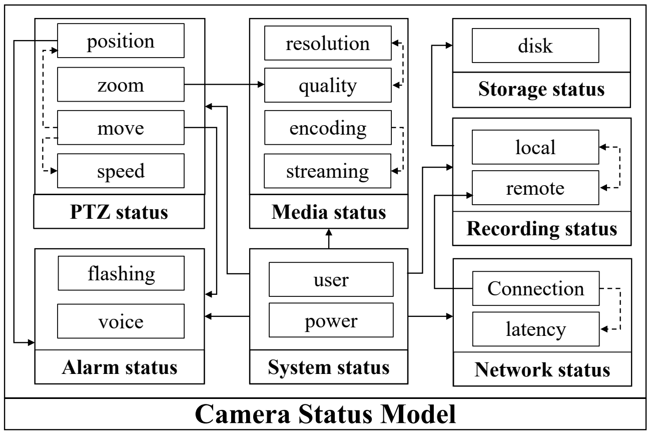


Figure 4. The constructed camera status model comprises seven statuses. Solid lines indicate internal dependencies within specific sub-statuses, while dotted lines represent inter-dependencies between different statuses.

- **Media Status**: Describes the characteristics of the video stream. The *resolution* and *quality* fields define the visual clarity of the footage; *encoding* specifies the compression format (e.g., H.264); and *streaming* indicates whether the video feed is actively being transmitted.
- **Alarm Status**: Indicates the status of alarm mechanisms. It includes *flashing*, which controls visual alerts such as blinking LEDs, and *voice*, which manages audio warnings. These alarms are typically triggered by motion events or system rules.
- **Storage Status**: Reflects the availability of on-device storage resources, such as internal memory or attached disks. It indicates whether the storage is operational, full, or unavailable.
- **Recording Status**: Captures the current recording state of the device. The camera may record to either local or remote destinations, and this status includes modes such as *recording*, *idle*, or *finished*.
- **Network Status**: Represents the connectivity condition of the camera. It includes *connection*, which denotes whether the device is online, and *latency*, which measures communication delay or responsiveness.
- **System Status**: Covers global system attributes such as *user*, which stores information about the active user session, and *power*, which reflects the power state of the device (e.g., on, off, rebooting).

We formally define each status category in the model as S_i, and its corresponding sub-status fields as s_i, where $s_i \in S_i$. When the value of a sub-status changes, we denote the transition using the notation $s_i \to s'_i$, where s'_i represents the updated value of s_i. A change in any sub-status s_i implies that the overall status S_i has also changed. Formally, we define a status transition as follows:

$$\forall s_i \in S_i,\ s_i \to s'_i \implies S_i \to S'_i$$

The edges in the camera status model represent dependencies between different statuses or sub-statuses. We classify these edges into two categories: The first is the inner edge, which represent intra-status dependencies—that is, relationships between sub-statuses within the same status group. An inner edge indicates that a change in one sub-status may trigger or imply a change in another sub-status from the same S_i. Formally, an inner edge is defined as:

$$E\_inn(s_i, s_j) : s_i, s_j \in S,\ s_i \to s'_i \Rightarrow s_j \to s'_j$$

This means that if any change occurs in a sub-status s_i, and another sub-status s_j, which belongs to the same status S, always changes as a result, then there exists an *inner edge* between s_i and s_j.

For example, within the PTZ status, when the sub-status `move` changes (e.g., from idle to rotating), the sub-status `position` is also updated accordingly. This reflects a causal dependency captured by an inner edge. Beyond inner edges, the second type of edge is an *inter edge*, which captures dependencies across different status categories. Based on the origin and destination of the edge, we further classify inter edges into two types. The first type of inter edge describes the dependency between two sub-statuses from different status groups. Formally, it is defined as:

$$E\_int(s_i, s_j) : s_i \in S_i,\ s_i \in S_j,\ s_i \to s'_i \Rightarrow s_j \to s'_j$$

Similar to the inner edge, this type of inter edge indicates that a sub-status s_j always changes when another sub-status s_i changes, but s_i and s_j belong to different status groups. For example, in the camera status model, when the sub-status `zoom` in the PTZ status is updated, it often triggers a change in the `quality` sub-status under the Media status, due to internal camera reconfiguration processes. This relationship is captured by an inter-sub-status edge. The second form of inter edge represents dependencies between entire statuses. It describes a situation where a change in any sub-status within S_i can affect the overall status S_j. This is formally defined as:

$$E\_int(S_i, S_j) : \exists s_i \in S_i,\ s_i \to s'_i \Rightarrow S_j \to S'_j$$

This means that a change in any sub-status within S_i leads to changes in all sub-statuses in S_j, indicating a strong inter-status dependency. For example, if the `power` field in the System Status changes (e.g., due to a shutdown or reboot), it typically affects all other functional statuses, such as halting PTZ movement, interrupting media streaming, or disabling alarms.

All edges in the camera status model are derived from empirical analysis of packet interactions between the camera and client applications. We first manually label each protocol packet with the primary status it directly affects. For instance, an HTTP `move-up` request is labeled as affecting the `move` sub-status within the PTZ status. Next, we analyze the response behavior and observe secondary effects. For example, we may find that when the `move` sub-status is updated, the `position` and `speed` sub-statuses also change in the subsequent responses. As a result, we add inner edges from `move` to `position` and `move` to `speed` to reflect these causal relationships. This modeling process allows us to systematically capture both explicit and implicit dependencies among camera functionalities, which are later used to guide interaction-aware fuzzing.

Although the labeling process and edge construction are performed manually, they are designed to be one-time efforts. The camera status model captures protocol-level behaviors and interactions that are consistent across a wide range of commercial IP cameras adhering to standard protocols such as ONVIF, HTTP, and RTSP. Once the model is constructed, it can be reused across different devices.

4.2.2. Testing Packets Generation. Leveraging the camera status model, `CAMVEIL` generates test packets that specifically target relevant status fields to explore protocol interactions. To bootstrap this process, `CAMVEIL` first collects protocol-compliant packets from real-world camera clients. After that, each one is labeled with the corresponding status or sub-status it attempts to access. All labeled packets are stored in a packet corpus for reuse. During each fuzzing round, `CAMVEIL` randomly selects one seed packet and then selects two additional packets from other protocols that either access the same status or are related via status dependencies defined in the camera status model. Each selected packet is then mutated in a type-aware manner. Specifically, `CAMVEIL` identifies protocol fields (e.g., integers, enumerations, strings) and applies mutations that preserve their type and structural validity, thereby increasing the chance of triggering deep logic in the camera while avoiding early rejection due to malformed formats.

For ONVIF packets, which use SOAP/XML encoding, `CAMVEIL` implements both value-level and structure-level mutations. At the value level, it mutates the textual content of XML elements to cover boundary values, protocol keywords, and syntactic anomalies. At the structure level, it introduces element reordering, optional field insertion/removal, and unexpected nesting to test the robustness of the camera's XML parser and SOAP handler.

In addition, to explore potential command injection or RCE vulnerabilities, such as those exemplified by CVE-2023-3959 in Section 3.1, `CAMVEIL` performs targeted semantic injections. For XML elements, HTTP query strings, or other string-type fields, the fuzzer mutates content using a curated set of payloads that include suspicious directives like `sleep=9999` and `reboot`. These payloads are inserted in contexts where they may be interpreted as embedded commands by vulnerable backend handlers, enabling the discovery of logic bugs beyond standard format violations.

4.3. Logic Vulnerability Monitor

To determine whether a test input triggers a functional vulnerability in the camera, we design a logic-level monitoring component. This component continuously probes the camera's runtime behavior by issuing status queries using standard protocol clients, HTTP, RTSP, and ONVIF. Each client is responsible for retrieving one or more of the seven statuses defined in the camera status model. All status requests and their corresponding responses are stored in a request-response pool in the form of key-value pairs, where the key is a timestamped request ID and the value is the associated response. This structured storage allows efficient lookup and comparison of camera states over time.

To evaluate whether abnormal behavior occurs, we deploy two specialized monitors: the *timeout monitor* and the *inconsistency monitor*. The timeout monitor detects cases where the camera becomes unresponsive to standard status queries within a predefined threshold, indicating potential service hangs or crashes. The inconsistency monitor compares the latest camera response with previously recorded results to identify logic-level anomalies such as status divergence, state corruption, or video feed freezing. The workflow of these two monitors is detailed in Algorithm 1.

Algorithm 1: The workflow of the logic monitor

```
Input  : $P_r$: The current req-res pool
         $R_i$: The request sent by the status getter
Output: $V_l$: The logic vulnerabilities detected
1  await $res$ = sendToCamera($R_i, timeout$);
2  if $res$ == null then
3      $V_l$.construct($R_i$, TIMEOUT); return $V_l$;
4  end
5  $responseInPool$ = fetchResponse($P_r, R_i$);
6  if $responseInPool$ == null then
7      $P_r$.add($R_i$, $res$);
8  end
9  $returnCode$ = $res$.code;
10 if $responseInPool.code$ == $returnCode$ then
11     originMessage =
         $responseInPool$.statusMessage;
12     if originMessage != res.statusMessage then
13         $V_l$.construct($R_i$, INCONSISTENT); return
             $V_l$;
14     end
15 end
16 else
17     $V_l$.construct($R_i$, INCONSISTENT); return $V_l$;
18 end
```

The inputs to Algorithm 1 are the current request-response pool P_r and the status query request R_i issued by one of the protocol-specific status getters. The output is a set of logic vulnerabilities V_i identified based on the camera's runtime behavior. As shown in line 1, `CAMVEIL` sends the request R_i to the target camera with a predefined timeout. If no response is received within this timeout window (lines 2–4), the monitor flags a potential timeout vulnerability by calling `construct` with the label `TIMEOUT`. This captures issues such as communication hangs or firmware stalls.

Next, in line 5, the algorithm attempts to retrieve a previously recorded response for the same request R_i from the pool P_r. If no prior response is found (line 6), the current request-response pair is stored (line 7), and the algorithm exits without reporting a vulnerability. If a historical response exists, the algorithm proceeds to compare it with the new response. As shown in lines 9–15, the first comparison is on the return code (e.g., HTTP status code, ONVIF result code). If the return codes differ (line 16), the inconsistency is flagged as a logic vulnerability. If the return codes match, the algorithm performs a finer-grained comparison by checking the actual `statusMessage` fields (lines 11–13). For example, two responses may both return code 200, but with different status messages such as `OK` versus `Partial Content`. Any mismatch in these messages is considered an inconsistency, and `CAMVEIL` reports it accordingly. This dual-level monitoring, based on both connectivity and semantic consistency, enables `CAMVEIL` to detect subtle logic bugs that do not cause crashes but still indicate abnormal or vulnerable behavior.

Handling False Positives. Our inconsistency detection relies on the response code and `statusMessage`, which, by design, reflect the semantic validity of a request rather than the device's current state. State-specific info (e.g., ”Moving” vs. ”Idle”) is typically in the response body or separate data fields. For example, a GetStatus command always returns 200 OK, but the body specifies whether the device is in the “Moving” or “Idle” state. Thus, under normal conditions, identical requests yield the same response code/`statusMessage`. Discrepancies are rare.

However, we observe that the logic oracles in `CAMVEIL` may occasionally produce false positives. For example, when an HTTP request is issued to move the camera upward, the initial response might be `200 OK`. However, if the camera reaches its mechanical upper limit, subsequent responses may change to `409 Conflict`. While this change reflects expected behavior, the oracle may incorrectly interpret it as an inconsistency. To mitigate such cases, we incorporate a manual verification step for all candidate vulnerabilities flagged by the monitor. Specifically, we identify request patterns that may produce variable but legitimate responses and verify whether the observed change reflects a true functional issue. This verification is performed across multiple camera states and device sessions to ensure consistency. If the anomaly consistently appears under controlled conditions and deviates from expected behavior, we classify it as a logic vulnerability and report it for further validation.

Bug Reproduce. To reproduce the logic vulnerabilities detected by `CAMVEIL`, we leverage the complete log of protocol messages recorded during fuzzing. Once a potential vulnerability is identified by the monitors, we first attempt to reproduce the issue by re-sending the most recent test packets from each involved protocol (e.g., ONVIF, HTTP, or RTSP). If the issue does not reappear, we perform a backward traversal through the previously sent messages, gradually replaying earlier sequences until the vulnerability is successfully triggered again. This step-wise reproduction process ensures the root cause packet or sequence is isolated and verifiable. To further validate the bug, we repeat the reproduction multiple times under the same network and camera state to confirm its stability, and rule out side effects caused by network latency or race conditions.

5. Implementation

We implement `CAMVEIL` using a combination of protocol parsing, traffic capture, and status-aware mutation techniques. For HTTP and RTSP packets, we adopt the Scapy library [16], which provides flexible support for deserialization and field-level mutation. However, since Scapy does not natively support ONVIF parsing, we implement a custom parser to handle the SOAP/XML structure [17] used in ONVIF requests and responses. To collect real-world protocol packets for status labeling and mutation, we interact with the camera using standard protocol clients. For HTTP, we use vendor-provided web interfaces accessed via a browser. For RTSP, we use the VLC media player [18], which streams live video and issues standard RTSP control messages. For ONVIF, we utilize `easy_onvif` [19], an open-source Dart implementation of the ONVIF protocol. These real-world packets form a diverse and representative corpus, allowing `CAMVEIL` to perform context-aware mutation across protocols, and effective multi-protocol fuzzing guided by our camera model.

To adapt `CAMVEIL` to another camera, no firmware access or device-specific customization is required. Since the camera status model is protocol-driven and protocol-agnostic at the packet level, it can be reused across different camera models that conform to standard protocols such as ONVIF, HTTP, and RTSP. To test a new security camera, we need to follow 4 steps: 1) Setup the camera and `CAMVEIL` in the same local area network. 2) Using `CAMVEIL` to capture the normal packets sent by HTTP, RTSP and ONVIF clients. 3) Label the statuses accessed by the collected normal packets. This step requires some human effort, but different cameras generally share similar types of protocol requests. For example, most cameras follow the ONVIF standard, meaning they support common requests like ‘get-services’ (accessing system status) and media ‘multi-casting’ (accessing media status). Similarly, most HTTP packets use a POST request with position information to move the camera. Therefore, most patterns labeled for one camera can often be reused for another, keeping the human effort in this step manageable. 4) Start the fuzzing process and checks whether any vulnerabilities occur.

6. Evaluation

6.1. Environment Setup

To set up the testing environment for security cameras, we deployed the cameras and `CAMVEIL` within the same local network. First, `CAMVEIL` captured 1,000 normal packets for each protocol. Based on these collected packets, we identified the camera statuses to monitor, then initiated the fuzzing process and observed the camera's behavior. For comparison, we also set up state-of-the-art fuzzing tools. For Peach, we deployed multiple instances, with each instance dedicated to fuzzing a single protocol independently. For DIANE, we attempted to replicate their evaluation setup by purchasing the devices they used. However, as many models are no longer commercially available, we were only able to obtain one: a FOSCAM camera of type FI9831P, which is supported by DIANE's companion app analysis approach.

We deployed CAMVEIL on a MacBook Pro (2019 model) to run all the evaluation tasks. The machine is equipped with a 64-bit Intel Core i7 processor (6 cores, 12 threads), running at 2.6 GHz. It has 16 GB of LPDDR4 memory and a 512 GB SSD for storage. All software components, including protocol parsers, mutators, and logic monitors, were run locally without requiring remote servers or distributed systems. This setup demonstrates that CAMVEIL can operate efficiently on widely available hardware and is capable of conducting both comprehensive testing and even real-world attacks.

TABLE 1. SUMMARY OF THE SECURITY CAMERAS UNDER TEST.

ID	Vendor	Model	Firmware Version
1	Hikvision	DS-2SC3Q140MY-TE	V5.7.20
2	Hikvision	DS-2CD2X22FWD	V5.3.0
3	Honeywell	HVCD-43001	V1.000.HW01.0.R
4	TP-Link	TL-IPC44K-4	1.0.1
5	TP-Link	TL-IPC44AW	1.0.12
6	TP-Link	TL-IPC433H-A4-W10	1.0.6
7	FOSCAM	F19831P	1.5.2.11
8	EZVIZ	H9c	V5.3.8
9	Santachi	ST-AD100	1.0.0

The detailed information of the security cameras we chose for evaluation can be found in TABLE 1. All devices were commercially available off-the-shelf products purchased from major online retailers. The selected cameras span nine different camera models from six well-known vendors, including Hikvision, Honeywell, TP-Link, FOSCAM, EZVIZ, and Santachi. These devices vary in terms of functionality, firmware implementations, and vendor ecosystems, and collectively represent a diverse and representative sample of widely deployed IP cameras in both consumer and enterprise environments. This diversity ensures that our evaluation results reflect real-world scenarios and highlights the generalizability of CAMVEIL across heterogeneous camera platforms.

While our approach bootstraps from 1000 intercepted packets, we repeated the experiments for various times and obtained consistent results. In each run, the initial packets were collected by exercising the full set of client operations (e.g., rotation, video streaming, triggering alarms), ensuring that the seed corpus covers the main protocol functionalities.

6.2. Vulnerabilities Found in Security Cameras

Totally, we have adapted CAMVEIL on 9 security cameras from 6 vendors, including Hikvision, Honeywell, TP-Link, FOSCAM, EZVIZ, and Santachi. These cameras were selected as evaluation targets due to their widespread deployment in both industrial and residential environments. Hikvision leads the IP surveillance market with a 20–25% global share [20], and models like DS 2SC3Q140MY TE and DS 2CD2X22FWD are part of its mainstream product lines. Besides, EZVIZ is also a top player in the smart home camera market [21]. Our selection spans 6 vendors to ensure codebase diversity. Currently, we have found 22 vulnerabilities in these devices. Details can be found in TABLE 2. Each fuzzing session was configured to run for 24 hours. In practice, however, most vulnerabilities were triggered within the first two hours of testing. Upon detecting a vulnerability, we immediately halted the current run, saved all relevant logs and packet traces for further analysis, and then rebooted both the camera and the fuzzing environment before continuing with the next round of testing.

Bug Types. As shown in the table, the identified vulnerabilities fall into three categories: The first and most common type is *Video Freezing*, observed in 16 of the discovered cases. These bugs manifest as streaming timeouts, causing the video feed in VLC players, HTTP web interfaces, or ONVIF clients to become unresponsive or frozen. This type of vulnerability is detected by the timeout monitor in CAMVEIL. The second type is *Inconsistent Response*, where identical test inputs produce inconsistent outputs, such as differing HTTP status codes or response content. CAMVEIL detected 4 such bugs across multiple devices using its inconsistency monitor. The final type is *Internal Error*, in which protocol interactions lead to disconnection or system instability. Two cases of this type were observed. This category aligns with bugs typically captured by crash monitors, as used in prior works like DIANE. CAMVEIL incorporates similar mechanisms to identify such failures.

Bug Severity. All the discovered vulnerabilities have the potential to cause significant security and operational consequences. For *Video Freezing* vulnerabilities, attackers can launch denial-of-service (DoS) attacks by remotely sending specially crafted protocol packets. These packets can freeze the live video feed accessed via VLC players, HTTP interfaces, or ONVIF clients, effectively disabling real-time surveillance. This allows attackers to operate undetected within the camera's coverage area, potentially leading to property loss or security breaches. For *Inconsistent Response* vulnerabilities, attackers may exploit inconsistencies in response handling to manipulate camera behavior. For instance, some malformed or invalid requests still receive `200 OK` responses (e.g., in Bug #3, #8, #10, and #11), enabling attackers to craft deceptive commands that bypass input validation and trigger unintended camera movements or configuration changes. Finally, *Internal Error* vulnerabilities can cause disconnection between the camera and its clients, effectively interrupting all real-time monitoring. Such vulnerabilities may be exploited to launch DoS attacks by destabilizing the device through protocol-level interactions, thereby disabling its surveillance functionality.

The affected vendors are listed as CNAs, so we cannot request CVEs directly from MITRE. CVE assignments must be initiated by the vendors. We are currently assisting them in the application process. Once approved, the CVEs will be published.

Bug Disclosure. We reported all the bugs to the vendors

TABLE 2. Vulnerabilities discovered by Camveil in cameras of Hikvision, Honeywell, TP-Link and FOSCAM.

Num	Vendor	Model Type	Bug Type	Bug Description	Report Time	Vendor Response	Patched Version
1	Hikvision	DS-2SC3Q 140MY-TE	Video Freezing	RTSP packets with CSeq over 256 bytes cause HTTP video freezing.	2025-06-17	2025-07-09 confirmed	/
2			Video Freezing	RTSP packets with url over 4096 bytes cause HTTP video freezing.	2025-06-17	2025-07-09 confirmed	/
3			Video Freezing	Same ONVIF packets sometimes get '200 OK', other times timeout.	2025-06-17	2025-07-09 confirmed	/
4		DS-2CD2X 22FWD	Video Freezing	HTTP packets with over-sized field may lead to RTSP crashes.	2025-06-17	2025-07-09 fixed	V5.8.1
5			Video Freezing	Global variable reused in multiple protocols and cause semantic errors.	2025-06-17	2025-07-09 fixed	V5.8.1
6	Honeywell	HVCD-43001	Video Freezing	RTSP packets with CSeq over 1024 bytes cause HTTP video freezing.	2024-10-21	2024-11-06 confirmed	/
7			Video Freezing	RTSP packets with url over 4096 bytes cause HTTP video freezing.	2024-10-21	2024-11-06 confirmed	/
8	TP-Link	TL-IPC44K-4	Inconsistent Response	Same HTTP get '200 OK' or '443 Entity Too Large' responses.	2024-10-09	2024-10-15 fixed	1.0.2
9			Internal Error	HTTP and ONVIF packets on related statuses lead to disconnection.	2024-10-15	2024-10-18 fixed	1.0.2
10		TL-IPC44AW	Inconsistent Response	Same HTTP get '200 OK' or '443 Entity Too Large' responses.	2024-10-09	2024-10-15 fixed	1.0.12(V6.0)
11		TL-IPC433H -A4-W10	Inconsistent Response	Same HTTP get '200 OK' or '443 Entity Too Large' responses.	2024-10-09	2024-10-15 fixed	1.0.8
12			Video Freezing	Media statuses accessed by RTSP are corrupted by HTTP packets.	2024-09-12	2024-09-24 fixed	1.0.8
13	FOSCAM	FI9831P	Video Freezing	RTSP packets lead to ONVIF video requests timeout and freezing.	2025-06-17	2025-08-23 further reported	/
14			Inconsistent Response	ONVIF ptz requests get '200' response and other times get errors.	2025-06-17	2025-08-23 further reported	/
15	EZVIZ	H9c	Video Freezing	RTSP with User-Agent exceeding 1900 causes the HTTP video freeze.	2025-06-17	2025-07-09 confirmed	/
16			Video Freezing	RTSP with Cseq exceeding 1900 causes the HTTP video freeze.	2025-06-17	2025-07-09 confirmed	/
17			Video Freezing	RTSP with invalid request headers causes the HTTP video freeze.	2025-06-17	2025-07-09 confirmed	/
18			Video Freezing	RTSP with URL path exceeding 1920 causes the HTTP video freeze.	2025-06-17	2025-07-09 confirmed	/
19			Video Freezing	RTSP with an invalid URL structure causes the HTTP video freeze.	2025-06-17	2025-07-09 confirmed	/
20			Video Freezing	RTSP DESCRIBE request with long Accept field leads to video freeze.	2025-06-17	2025-07-09 confirmed	/
21	Santachi	ST-AD100	Video Freezing	Interaction between RTSP and ONVIF freeze the video stream.	2025-06-17	2025-08-23 further reported	/
22			Internal Error	The network interface crashed after being tested under crafted packets.	2025-06-17	2025-08-23 further reported	/

immediately after preparing the reproduction script and collecting the bug manifestation photos or video. 18/22 of the bugs are acknowledged by the vendors in about 1-2 weeks. For the bugs without responses, we tried to further contact the vendors in about 1-2 months. 7 of the bugs are patched by the vendors, we listed the related versions in Table 2.

Bug Types. Among the 22 discovered bugs, 16 are memory bugs and 6 are logic bugs. All memory bugs are vulnerabilities, as they can cause overflows or leaks leading to crashes or RCE. Of the logic bugs, 2 are vulnerabilities that can crash the camera, while bugs #8, #10, #11, and #14 are not, as they only cause inconsistent responses without exposing an attack surface.

Bug Root Causes. The detected bugs mainly fall into two categories: 1) Memory check deficiency. Missing boundary checks on RTSP/HTTP fields (e.g., long CSeq, URL, or headers) caused buffer overflows, where the overflowed data interfered with adjacent protocol handlers. This root cause accounts for 15 out of 22 bugs. 2) Global variable concurrency conflict. Different protocols (HTTP, RTSP, ONVIF) concurrently accessed shared global variables without proper synchronization, leading to inconsistent responses or stream corruption. This root cause explains 7 out of 22 bugs. Camveil's cross-protocol fuzzing was essential in surfacing these vulnerabilities, as they would not have been exposed under single-protocol testing.

Bugs found by other tools. We also evaluated two state-of-the-art fuzzing tools, Peach and DIANE, on the same set of devices. Notably, DIANE supports only the FOSCAM camera due to its hardware dependencies. The result shows Peach successfully detect Bug #22, while DIANE detects no bugs. The primary reason is that most of the discovered bugs (excluding Bug #9 and Bug #22) do not result in device crashes. Both Peach and DIANE are primarily designed to detect crash-based vulnerabilities, and thus failed to capture these logic-level issues. To further evaluate the contribution of our camera status model, we extended Peach by integrating the same logic monitors used in `CAMVEIL`. With this enhancement, Peach was able to detect 4 bugs (Bug #8, #10, #11, and #22). However, it still missed the remaining bugs, which require coordinated interaction across multiple protocols to be triggered. These results highlight the importance of status-aware, cross-protocol test generation. Without sending interrelated messages guided by the camera status model, such complex vulnerabilities remain undetected.

Ablation Study. To evaluate the contribution of each core component in `CAMVEIL`, we conducted an ablation study by selectively disabling key modules and observing the impact on bug detection.

We first disabled the Camera Status Model, resulting in protocol-agnostic random fuzzing. In this configuration, `CAMVEIL` sent isolated, randomly selected packets under a non-coordinated setting, where messages are generated independently without modeling cross-protocol dependencies. . As a result, only 3 bugs (Bug #6, Bug #8, Bug #9) were discovered. All other bugs, which require protocol interaction, remained undetected. Next, we disabled the

Logic Vulnerability Monitor, relying solely on crash-based detection (e.g., segmentation faults, TCP disconnections). In this setup, only Bug #9 and Bug #22 (two internal errors causing disconnection) was detected. The remaining 20 logic bugs, which do not result in crashes, were entirely missed.

These results demonstrate that both components are essential: the status model enables semantically meaningful test generation, while the logic monitor uncovers impactful vulnerabilities beyond crash bugs.

False Positives. During our experiments, we observed two false positives, resulting in a false positive rate of $2/(22+2) = 8.33\%$. Both false positives occurred during testing on TP-Link cameras. In these cases, the initial test request received a `200 OK` response, while the subsequent identical request returned a `409 Conflict`. Upon manual inspection, we confirmed that these were not true vulnerabilities but instead expected behavior: the camera had reached its physical rotation limit and could not process further pan or tilt commands.

Reproducing Bugs. CAMVEIL successfully reproduced 20 out of 22 reported bugs, yielding a 90.9% success rate. Bugs #9 and #22 could not be reproduced due to their reliance on precise timing. For example, bug #9 requires modifying memory read by HTTP exactly as ONVIF updates a buffer, while bug #22 involves a shared global variable being altered within a critical window.

6.3. Typical Case Study

To illustrate how CAMVEIL identifies vulnerabilities in practice and to provide deeper insights into their root causes, we present four representative cases for in-depth analysis. These case studies highlight the diverse types of logic vulnerabilities uncovered and demonstrate the effectiveness of our multi-protocol coordinated fuzzing approach. Cases 1–4 arise from two protocols concurrently accessing the same memory region, causing read/write conflicts. The root causes include memory overflow (cases 1&2) and global variables concurrent accessing (cases 3&4). These bugs were exposed only by CAMVEIL, which intentionally drives cross-protocol interactions to access shared states, revealing memory conflicts. In contrast, existing fuzzers operate on isolated protocols and cannot trigger such issues.

Case 1: The first case involves a vulnerability we discovered in a TP-Link camera, model TL-IPC433H-A4-W10, listed as bug #12 in TABLE 2. As shown in the Fig 5, we initiated CAMVEIL to fuzz the camera within the same local network provided by a router. During testing, the media stream in the VLC player froze unexpectedly: the terminal time displayed 09:44:45, while the VLC video stream time remained at 09:43:51, indicating a media transmission freeze. We further observed that the video stream remained frozen indefinitely, and playback only resumed after manually rebooting the camera. Analysis revealed that this issue was triggered when CAMVEIL sent a combination of HTTP and RTSP packets that concurrently accessed and interfered with the camera's media status. The cross-protocol interference caused the internal media pipeline to become unresponsive. This vulnerability has been acknowledged and subsequently patched by the vendor.

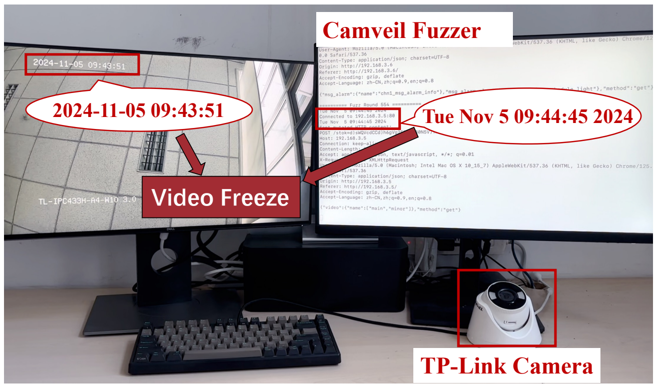


Figure 5. A real vulnerability case found by Camveil in the TP-Link camera. The terminal time displayed 09:44:45, while the video stream time remained at 09:43:51, indicating a media transmission freeze.

To further investigate the root cause of this vulnerability, we contacted the vendor and obtained the corresponding firmware for the affected TP-Link camera. Through reverse engineering and static analysis of the firmware binary, we identified the relevant code snippet shown in Figure 6.

```
char *__fastcall curl_maprintf(int a1, int a2, int a3, int a4){
  ...
  v4 = sub_104D8((int)&ptr, sub_10444, a1, (int *)varg_r1) == -1;
  ...
}
```

```
int __fastcall sub_10444(unsigned __int8 a1, int a2){
  ...
  v8 = ((void *(__fastcall *)(void *, size_t))realloc_0)(v3, 2 * v7);
  if ( !v8 )
    goto LABEL_3;
  ...
LABEL_3:
  result = -1;
  *(_DWORD *)(a2 + 12) = 1;
  return result;
```

allocation failed

not free
the memory

Figure 6. Code snippets from the vulnerability case in the TP-Link camera. HTTP packets corrupt the memory, and subsequent RTSP packets attempt to access this corrupted memory, causing the system to become unresponsive. Camveil triggers this bug by coordinating the HTTP and RTSP packets that access the same fields.

As shown in Figure 6, the malformed HTTP packet triggers the execution of the `curl_maprintf` function, which in turn invokes `sub_10444`. Within this function, a memory allocation is attempted via `realloc_0`. However, due to field mutations introduced by the fuzzed HTTP packet, the allocation may fail. When this happens, the variable `v8` is set to `NULL`, causing the program to jump directly to `LABEL_3`. This bypasses any memory cleanup or error handling, resulting in a memory leak and leaving internal state inconsistent. Subsequently, when an RTSP test packet, such as a `PLAY` request, is sent, the RTSP handler accesses the corrupted media status, which leads to the observed freeze in the real-time video stream. This vulnerability is difficult for existing fuzzing tools to detect, as it does not cause a system crash or disconnection. Instead, it arises from a subtle cross-protocol interaction where one protocol corrupts internal state, and another protocol subsequently triggers the faulty behavior. CAMVEIL detected such issues by multi-protocol coordination and logic-level monitoring.

Vulnerability's Severity and Exploitation. This vulnerability poses a security risk, as it enables attackers to launch denial-of-service (DoS) attacks that disrupt the camera's real-time video functionality. Once triggered, the camera becomes unresponsive to RTSP streaming requests, effectively cutting off all live video feeds. Exploitation requires only local network access to the camera, which can be obtained through common techniques such as ARP spoofing or routing attacks [22]. Given the prevalence of IP cameras in both industrial surveillance and residential security systems, such vulnerabilities can have wide-ranging consequences, from surveillance blind spots to physical security breaches. The issue has been acknowledged and addressed by the vendor in a subsequent firmware update, mitigating the attack vector in newer device versions.

Case 2: The second case involves a vulnerability discovered in a Honeywell camera (model HVCD-43001), listed as Bug #7 in Table 2. In this scenario, CAMVEIL simultaneously sends a mutated RTSP OPTIONS packet and an HTTP request, both of which access the camera's media status. As a result, both VLC clients streaming and the web-based HTTP control interface become unresponsive. The camera stops serving RTSP responses, and the HTTP interface times out, indicating a failure in the shared media subsystem triggered by concurrent protocol interaction.

Unfortunately, we were unable to obtain the firmware for this camera model, so our analysis relies solely on the observed network behavior. We found that the vulnerability could be consistently reproduced by sending an RTSP packet with an excessively long URI, exceeding 4096 characters, starting with a standard RTSP stream path. At the same time, a mutated HTTP request containing a similarly long string in the URI field was sent, targeting the camera's video stream endpoint. Figure 7 shows the pair of packets used to trigger this issue. After repeated testing, we hypothesize that the camera experiences a memory handling issue, likely a buffer overflow or memory corruption, when parsing these overly long URI fields across different protocol handlers. The simultaneous access to the camera's media subsystem via both RTSP and HTTP seems to compound the issue, resulting in the complete unresponsiveness of both interfaces.

This vulnerability went undetected by existing tools during our evaluation. DIANE does not support this Honeywell model, and Peach, despite being equipped with our logic vulnerability monitor, was unable to uncover the bug. This is because the vulnerability requires a specific combination of malformed RTSP and HTTP packets with long URIs, whereas existing tools typically fuzz each protocol independently. In contrast, CAMVEIL's multi-protocol coordination and status-aware test generation were key to successfully identifying this issue.

Vulnerability's Severity and Exploitation. This vulnerability can be exploited to perform DoS attacks against the affected camera. In addition to freezing the real-time video stream, it also renders the HTTP control interface unresponsive, effectively disabling both live monitoring and remote management functionalities. An attacker requires only local network access to execute the attack. By disrupting both the streaming and control channels, attackers could create blind spots in surveillance systems, enabling further malicious activities without detection. The vulnerability has been acknowledged and addressed by the vendor in a subsequent firmware update.

```
OPTIONS rtsp://192.168.31.64:554//Streaming/Channels/101Aa0Aa1Aa2(>4096)
RTSP/1.0
CSeq: 1
User-Agent: VLC media player (LIVE555 Streaming Media v2010.02.10) ONVIF packet
```

```
GET /video_stream/110283A78F8d(>4096) HTTP/1.1
Host: 192.168.31.64
Authorization: Basic base64-encoded-credentials
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7)
Accept: application/json, text/html, */*                    HTTP packet
```

Figure 7. The packets used to trigger the vulnerability in HVCD-43001. Camveil sends a RTSP and HTTP request simultaneously, which both access the media status.

Case 3: The third case involves a vulnerability found in the FOSCAM FI9831P camera, listed as Bug #13 in Table 2. In this scenario, the ONVIF client becomes unable to receive video data, and the video stream freezes entirely. As shown in Figure 8, the client detects this failure and displays an on-screen warning labeled NO SIGNAL. This issue arises when CAMVEIL sends a combination of ONVIF and RTSP test packets, both of which interact with the camera's media streaming status. The cross-protocol interference appears to corrupt internal stream management logic, resulting in the loss of video transmission. During our evaluation, neither Peach (even when equipped with our custom logic monitors) nor DIANE was able to identify this vulnerability. Peach generated ONVIF and RTSP packets independently, without protocol-level coordination. However, this bug is only triggered by a specific sequence of interleaved ONVIF and RTSP messages, demonstrating the necessity of cross-protocol fuzzing for such logic vulnerabilities.

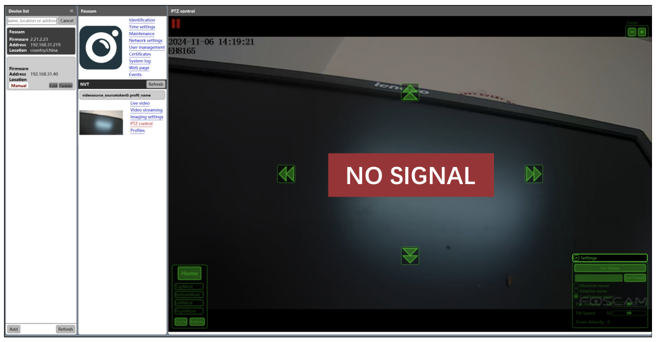
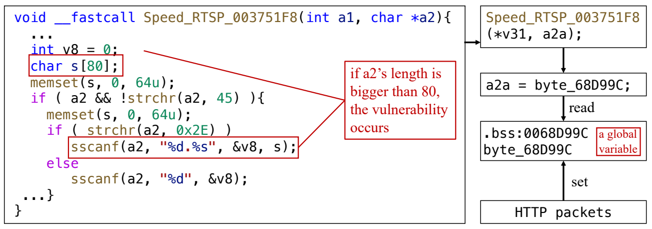


Figure 8. A real vulnerability case found by CAMVEIL in the FOSCAM camera. There is an alarm with the messages 'NO SIGNAL' in the middle of the screen

Vulnerability's Severity and Exploitation. Similarly, this vulnerability can be exploited to launch DoS attacks. In this case, not only is the real-time media stream disrupted, but the ONVIF Device Manager (ODM) interface also becomes unresponsive, effectively freezing both video output and camera control functionalities. An attacker only needs access to the camera's local network to launch the attack. By freezing both the control interface and the media output, the attacker not only disrupts live surveillance but also disables administrative recovery mechanisms via ONVIF. This elevates the severity of the attack, as even remote operators or automated recovery systems relying on ONVIF cannot restore the camera's functionality without a physical reboot or secondary access channel.

Case 4: The final case involves a vulnerability we discovered in a Hikvision camera, model DS-2CD2X22FWD, listed as Bug #4 in TABLE 2. During testing, we observed that RTSP packets received no response, and the video stream became frozen. This issue was triggered by coordinated testing involving both RTSP and HTTP protocols. To investigate the root cause of the bug, we attempted to analyze the camera's firmware. However, we found that the firmware was encrypted, preventing direct reverse engineering. Fortunately, we discovered a blog post [23] that introduced a method to decrypt Hikvision firmware, specifically including the one used by our target device. The blog outlines a technique involving the extraction of the decryption key from the bootloader and use of XOR-based decoding, allowing the firmware image to be unpacked for further analysis. This enabled us to proceed with static analysis and locate the relevant code associated with the vulnerability. The related code snippet is shown in Figure 9.



```
void __fastcall Speed_RTSP_003751F8(int a1, char *a2){
  ...
  int v8 = 0;
  char s[80];
  memset(s, 0, 64u);
  if ( a2 && !strchr(a2, 45) ){
    memset(s, 0, 64u);
    if ( strchr(a2, 0x2E) )
       sscanf(a2, "%d.%s", &v8, s);
    else
       sscanf(a2, "%d", &v8);
 ...}
}
```

```
Speed_RTSP_003751F8
(*v31, a2a);
```

```
a2a = byte_68D99C;
```

```
.bss:0068D99C
byte_68D99C
```

```
HTTP packets
```

Figure 9. Code snippets from the vulnerability case discovered by CAMVEIL in the Hikvision camera. HTTP packets set a global variable, and subsequent RTSP packets attempt to access this variable, causing the system to become unresponsive.

As shown in Figure 9, the malformed HTTP packet triggers an update to a global variable located at memory address 0x0068D99C. This global variable stores a string value that is later accessed by the RTSP handling function Speed_RTSP_003751F8. Specifically, the RTSP handler retrieves the string pointer a2a from this global variable and attempts to parse it using the function sscanf. Within Speed_RTSP_003751F8, the function first declares a fixed-size buffer char s[80] and then conditionally parses the value of a2 using the format string "%d.%s" into an integer and the buffer s. However, due to mutations introduced by the fuzzed HTTP packet, the string assigned to the global variable may exceed 80 bytes. If the RTSP packet subsequently triggers this vulnerable path, the sscanf operation overflows the buffer s, resulting in a memory corruption that causes system unresponsive.

This vulnerability is particularly challenging for conventional fuzzers to detect, as it requires a coordinated sequence of cross-protocol interactions: an HTTP packet must first set a global variable with an oversized string, and an RTSP packet must then invoke a vulnerable parsing function that misuses this global value. Such inter-protocol dependencies are typically outside the scope of single-protocol fuzzing tools, highlighting the necessity of CAMVEIL's coordinated multi-protocol approach.

Vulnerability's Severity and Exploitation. This vulnerability introduces a DoS risk by allowing remote attackers to corrupt memory through cross-protocol interaction. Beyond causing a system freeze, this memory corruption could potentially be escalated. If the attacker carefully crafts the injected string, it may overwrite return addresses or function pointers on the stack, paving the way for control-flow hijacking. This opens the door to arbitrary code execution, such as injecting shellcode or spawning a reverse shell, depending on the surrounding memory layout and runtime protections. The attack can be launched by sending a specially crafted HTTP packet that sets an oversized global string variable, followed by an RTSP request that activates the vulnerable parsing routine. Because the malformed string persists in global memory across protocols, even short-lived access can result in persistent system-level failure. Given the widespread deployment of IP cameras in critical environments, the possibility of turning a cross-protocol parsing flaw into a persistent compromise highlights the importance of multi-protocol-aware fuzzing and secure memory handling in embedded systems.

7. Discussion

Generalizability and the scope. The camera status model proposed in this paper is designed to be general and applicable to a wide range of IP cameras, as most devices implement the core statuses and inter-status relationships captured in the model. In our evaluation, all six tested cameras adopted the same model without modification. However, certain edge cases may require model customization. This occurs in two scenarios: 1) Some cameras may not support specific functionalities. For example, cameras lacking motorized control will not include PTZ-related statuses such as *position* or *speed*. In such cases, the model should be pruned by removing the unsupported statuses and their associated dependency edges. 2) Some cameras support additional statuses. Other cameras may introduce new functionalities not covered by the original model. For instance, models with audio recording may require the addition of a *volume* or *microphone* status. These extensions involve adding new status nodes and evaluating whether their changes affect or are affected by existing statuses, such that appropriate dependency edges can be incorporated. Even if the camera model is only partially aligned with a specific device, fuzzing can still proceed. However, without an accurate status map, certain cross-protocol interactions may be missed, leading to reduced coverage and effectiveness.

For the generalizability of other protocols, we currently focused on HTTP, RTSP, and ONVIF because they are the dominant configuration in IP cameras (for example, ONVIF alone covers 72% of the global network video surveillance market [24]). However, Camveil's design is protocol-agnostic and can be extended by modeling dependencies. For example, we also tested a camera from Xstrive which support WebRTC, and uncovered a previously unknown bug: RTSP cooperated with the WebRTC testing packets triggered a freeze in the video stream [25].

For encrypted communication (e.g., HTTPS or RTSP over TLS), `CAMVEIL` cannot directly fuzz encrypted traffic in our current setup due to missing private keys, and thus cannot perform effective mutation on the ongoing packets. In future work, we plan to intercept decrypted traffic inside the client application (e.g., via reverse engineering or dynamic hooking), enabling effective fuzzing over encrypted channels. However, this requires protocol-specific client instrumentation and presents engineering challenges.

Some manufacturers employ proprietary protocols, often encapsulated within encrypted SDKs, which pose significant challenges for analysis. While proprietary protocols are less common in the broader market, they are present in certain specialized or high-security environments. The lack of public documentation and the use of encryption in these protocols hinder the ability to perform effective fuzz testing and vulnerability assessment. Reverse engineering such protocols is not only time-consuming but may also raise legal and ethical concerns. Given these constraints, `CAMVEIL` currently focuses on cameras utilizing standard protocols, ensuring broad applicability and compliance with legal standards. Future work may explore methods to ethically extend support to devices with proprietary protocols, potentially through collaboration with manufacturers or the development of generic analysis frameworks that can adapt to undocumented interfaces.

`CAMVEIL` also doesn't support non-camera systems like drones or robots yet, as they involve more complex protocol interactions. To extend support, we would need to build richer status models and integrate with protocols like MAVLink or DDS. For example, when adapting `CAMVEIL` to drones, the status model would need to incorporate flight-related states such as flight mode (AUTO, GUIDED), GPS fix level, and arming status.

More monitors. `CAMVEIL` integrates the crash monitor proposed by prior works [5], [6], and introduces a logic vulnerability monitor to detect non-crashing issues such as response timeouts and inconsistencies. However, not all logic vulnerabilities in security cameras manifest through such symptoms. In some cases, the camera responds correctly and consistently to protocol messages, yet fails to perform the expected physical behavior. For example, an ONVIF packet that instructs the camera to pan may return a `200 OK` response, while the camera does not move at all. Similarly, an HTTP request that sets the movement speed to level 5 may succeed syntactically, but the camera still rotates at a noticeably low speed.

These types of bugs are particularly difficult to detect using standard request-response monitors. One potential solution is to maintain a reference model that records the expected camera status transitions in response to each packet. By comparing the actual observed status with the predicted outcome, discrepancies can be identified as potential logic bugs. However, implementing such a model presents significant challenges. First, it is difficult to precisely define and update all affected statuses for each incoming packet, especially those that induce continuous or dynamic changes. For instance, a 'move-up' request may alter both the *position* and *speed* statuses over time, requiring fine-grained and real-time status tracking. Second, concurrency introduces ambiguity: when multiple conflicting commands are sent in parallel, (e.g., setting speed to 3 via HTTP and 5 via ONVIF), it becomes unclear which effect should take precedence, making deviation detection unreliable. Addressing these challenges requires more advanced semantic modeling and device-level feedback collection, which we leave as an important direction for future work.

Manual Effort and Scalability The manual effort in `CAMVEIL` mainly comes from two aspects: 1) Labeling the initial packet corpus with the corresponding status/substatus. Our labeling process begins by executing all available client commands (e.g., "pan left", "zoom in", "get bitrate", etc.) through the camera client. This produces around 50–100 packets covering different functional categories of the device. We then manually annotate these representative request–response pairs. After that, subsequent similar packets can be auto-labeled based on shared structure and semantics. For instance, once 'pan left' is labeled, commands like 'pan right' or 'tilt up' follow the same format and can be auto-labeled. Adapting `CAMVEIL` to a new camera only requires repeating this process and typically takes 1–2 hours. For example, I adapted `CAMVEIL` to a new camera from Xstrive, which takes me 35 minutes.

2) Verifying Candidate Bugs. Candidate bug verification requires manual effort to confirm whether an inconsistency reflects a real bug. In our experiments, `CAMVEIL` found 6 inconsistencies(indicating a low candidate count), 4 of which were true bugs. Verification involves: (1) replaying the triggering sequence, (2) checking whether the inconsistency persists, and (3) examining logs or device behavior. Each case typically takes under 30 minutes, making the overall manual effort modest.

For the scalability, the manual effort required by Camveil is modest and does not pose a barrier to scaling the approach across other devices. Since initial labeling can be largely reused through structural similarity, adapting to a new camera is typically a one-time effort of under 1–2 hours. Bug verification is also lightweight, given that Camveil produces only a small number of high-quality candidate inconsistencies. Moreover, as more packet templates and verified cases accumulate, the adaptation cost further decreases due to higher coverage of reusable patterns.

Peach's multi-protocol messages `CAMVEIL` is designed to automatically generate lots of concurrent message pairs that access the same camera status, thereby inducing conflicting interactions. This automated construction enables systematic exploration of cross-protocol race conditions without manual effort. In contrast, with Peach, generating such message pairs requires a scheduler to set up multiple Peach instances, each hardcoded with specific message types. For example, here we gave configurations (Pit files) of two Peach instances: Instance A sends RTSP PLAY packets, while Instance B sends ONVIF packets to set the video encoder configuration. Running these two fuzzers in parallel only tests one specific pair of concurrent accesses: RTSP and ONVIF both modifying the camera's media status.

To test any new pair of conflicting status accesses, users must create a new set of configurations, and restart the Peach instances to enable the new protocol pairs. In CAMVEIL, each camera status involves about 20 types of the packets, and thus producing about 20 * 20 = 400 concurrent message pairs. For Peach, we need to write about 2*30 (round code lines for each Pit file) * 400 (pairs) = 24,000 lines to config all these message pairs, which needs huge effort.

8. Related Work

Protocol Fuzzing. Many works have been conducted on developing fuzzing frameworks for protocol implementations. These approaches can generally be categorized into two types based on how they generate test cases: generation-based and mutation-based. The first category is generation-based protocol fuzzers [7], [8], [26], [27], [28], [29]. They create test inputs from scratch using protocol specifications. For example, Peach [7] is a widely adopted generation-based fuzzer that uses XML-formatted documents to define a protocol's state model and data structure. Based on this specification, Peach can synthesize protocol-compliant packets to explore different code paths systematically. Another representative tool is Defensics [30]. Defensics is a commercial fuzzing suite that provides support for hundreds of network protocols with millions of pre-defined test cases, enabling broad testing coverage without requiring manual input. Besides, Boofuzz [31] is another open-source framework of this type. It provides programmable APIs to define protocol states and fields.

The second category consists of mutation-based protocol fuzzers [32], [33], which generate test cases by mutating existing protocol messages captured from real executions. For example, AFLNet [32] extends the widely-used AFL [34] fuzzer to support stateful network protocols by mutating recorded packets and applying state-aware scheduling strategies. A more recent tool, ChatAFL [35], enhances this approach by integrating large language models to guide structure-aware mutations. Built on top of AFLNet, ChatAFL improves mutation effectiveness by better preserving protocol semantics during fuzzing. Another representative work is Bleem [8], which functions as a transparent proxy to capture live network traffic and perform real-time mutation. Unlike traditional fuzzers that focus on either server or client endpoints, Bleem supports fuzzing both sides of a communication session, making it applicable to diverse scenarios.

The work by Garousi et al. [36] leverages model-based testing (MBT) for end-to-end test automation of several large web and mobile applications. Similarly, the work by Godoy et al. [37] discusses search-based testing (SBT), in which they exploit a particular abstraction of object protocols to find failures. Our approach can be considered a specific form of MBT and search-based testing SBT. Like MBT, we use an abstract status model to guide the generation of valid, state-dependent fuzzing inputs. And like SBT, we explore sequences that can lead to unexpected inter-protocol interactions, though our "search" is guided by protocol-aware coordination rather than heuristic optimization.

IoT Device Fuzzing. In addition to protocol fuzzers, numerous efforts have focused on developing fuzzing techniques specifically tailored for IoT devices. One line of research targets the firmware of IoT devices [38], [39], [40], where the firmware is emulated or instrumented to enable dynamic fuzzing without requiring access to physical hardware. Another approach focuses on the mobile companion applications that communicate with IoT devices. For instance, IoTFuzzer [5] performs fuzzing by identifying and reusing device-specific logic (e.g., encryption routines) embedded within mobile apps to craft test messages. Building on this idea, DIANE [6] leverages static analysis and dynamic fuzzing to extract mutation points from Android apps, enabling field-level modifications before messages are sent to devices. Wang et al. [41] further extend this idea by using the companion app to infer device functionality and then extrapolate potential vulnerabilities based on known issues from similar devices. More recently, RIoTFuzzer [42] proposes a hybrid approach that combines app analysis with side-channel-guided fuzzing. By monitoring side-channel information, it enhances the effectiveness of fuzzing even when internal device states are opaque.

Main Differences. Unlike traditional protocol fuzzing tools, CAMVEIL performs multi-protocol coordinated fuzzing guided by a status-aware model, while existing fuzzers typically test each protocol in isolation. Compared to IoT device fuzzers such as DIANE, which rely on companion app modification, CAMVEIL directly mutates protocol-level packet fields, enabling fine-grained and systematic exploration of device behavior. Moreover, CAMVEIL integrates both logic vulnerability monitors and crash-based detectors, allowing it to uncover a broader spectrum of issues, including both semantic inconsistency bugs and timeout bugs, that are often missed by prior tools.

9. Conclusion

In this paper, we present a tool called CAMVEIL to uncover vulnerabilities in security cameras through multi-protocol coordinated fuzzing. By constructing a camera status model, CAMVEIL identifies key resources within a security camera and maps the dependencies among them. Utilizing the mode, CAMVEIL generates inter-related packets from multiple packets for camera testing. Additionally, to detect non-crash vulnerabilities, CAMVEIL includes a semantic oracle to check for response timeouts or inconsistencies from standard clients. We have successfully adapted CAMVEIL for 9 industrial cameras from Hikvision, Honeywell, TP-Link, FOSCAM, EZVIZ, and Santachi, detecting 22 critical vulnerabilities in these devices. In future ‘work, we aim to develop additional monitors to detect a broader range of vulnerabilities in security cameras under black-box conditions. We also plan to expand our testing to include more cameras from different vendors.

10. ACKNOWLEDGEMENTS

We would like to express our appreciation to the anonymous reviewers for their valuable and constructive feedback, as well as to the shepherd for the guidance in improving the paper. This research is sponsored in part by the National Key Research and Development Project (No. 2022YFB3104000), NSFC Program (No. 62502254, 62525207, U2441238) and China Postdoctoral Science Foundation (No. 2024M761690, GZC20240828).

References

[1] ONVIF, “Onvif is an open industry forum that provides and promotes standardized interfaces for effective interoperability of ip-based physical security products.” https://www.onvif.org/, 2024, accessed at October 11, 2024.

[2] D. Gourley and B. Totty, *HTTP: the definitive guide.* ” O’Reilly Media, Inc.”, 2002.

[3] H. Schulzrinne, A. Rao, and R. Lanphier, “Real time streaming protocol (rtsp),” Network Working Group, Tech. Rep., 1998.

[4] N. V. Database, “Cve-2023-3959 detail,” https://nvd.nist.gov/vuln/detail/CVE-2023-3959/, 2023, accessed at October 11, 2024.

[5] J. Chen, W. Diao, Q. Zhao, C. Zuo, Z. Lin, X. Wang, W. C. Lau, M. Sun, R. Yang, and K. Zhang, “Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing.” in *NDSS*, 2018.

[6] N. Redini, A. Continella, D. Das, G. De Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, “Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices,” in *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 2021, pp. 484–500.

[7] GitLab, “Peach is a smartfuzzer that is capable of performing both generation and mutation based fuzzing.” https://peachtech.gitlab.io/peach-fuzzer-community/, 2024, accessed at October 11, 2024.

[8] Z. Luo, J. Yu, F. Zuo, J. Liu, Y. Jiang, T. Chen, A. Roychoudhury, and J. Sun, “Bleem: Packet sequence oriented fuzzing for protocol implementations,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4481–4498.

[9] P. Kumar, A. Dick, and T. S. Sheng, “Real time target tracking with pan tilt zoom camera,” in *2009 Digital Image Computing: Techniques and Applications*, 2009, pp. 492–497.

[10] Google, “The device access program allows users to access, control, and manage google nest devices using the sdm api.” https://developers.google.com/nest/device-access/get-started, 2024, accessed at October 11, 2024.

[11] ONVIF, “The global security market goes for onvif,” https://www.onvif.org/pressrelease/the-global-security-market-goes-for-onvif/, 2025, accessed at May 11, 2025.

[12] ——, “Onvif reaches new milestone of 30,000 conformant products,” https://www.onvif.org/blog/2024/09/10/onvif-reaches-new-milestone-of-30000-conformant-products/, 2024, accessed at October 11, 2024.

[13] C. Huitema, “Real time control protocol (rtcp) attribute in session description protocol (sdp),” Network Working Group, Tech. Rep., 2003.

[14] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, “Rfc3550: Rtp: A transport protocol for real-time applications,” 2003.

[15] IOTSec-Zone, “Cve-2023-3959,cve-2023-4249 - zavio ip,” https://www.iotsec-zone.com/article/423, 2025, accessed at September 15, 2025.

[16] Scapy, “Welcome to scapy version 2.6.0,” https://scapy.net/, 2024, accessed at November 6, 2024.

[17] W3C, “Soap,” https://www.w3.org/TR/soap/, 2025, accessed at September 17, 2025.

[18] jb kempf, “15 years of vlc and videolan,” https://jbkempf.com/blog/15-years-of-VLC/, 2025, accessed at September 17, 2025.

[19] faithoflifedev, “Dart implementation of onvif ip camera client,” https://github.com/faithoflifedev/easy_onvif_workspace, 2024, accessed at November 6, 2024.

[20] FMI, “Cctv camera market demand & sustainability trends 2025-2035,” https://www.futuremarketinsights.com/reports/cctv-camera-market, 2025, accessed at September 15, 2025.

[21] T. Insight, “Insight: Ezviz and ring remain smart home surveillance camera leaders,” https://www.techinsights.com/blog/insight-ezviz-and-ring-remain-smart-home-surveillance-camera-leaders, 2025, accessed at September 15, 2025.

[22] J. Kristoff, “Local network attacks,” https://www.first.org/resources/papers/tc-oct2002/d1-s2-kristoff-slides.pdf, 2024, accessed at November 6, 2024.

[23] S. null latest developers news, “Hik or hack? (not) iot security using hikvision ip cameras,” https://sudonull.com/post/64821-Hik-or-hack-NOT-IoT-Security-Using-Hikvision-IP-Cameras-\\NeoBIT-Blog, 2017, accessed at May 11, 2025.

[24] ONVIF, “The global security market goes for onvif,” https://www.onvif.org/pressrelease/the-global-security-market-goes-for-onvif/, 2024, accessed at September 15, 2025.

[25] Camveil, “The bug lead to video freeze of webrtc,” https://anonymous.4open.science/r/CamVeil-15FB/xstrive-case.md, 2025, accessed at September 15, 2025.

[26] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, “Ics protocol fuzzing: Coverage guided packet crack and generation,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–6.

[27] J. Yu, Z. Luo, F. Xia, Y. Zhao, H. Shi, and Y. Jiang, “Spfuzz: Stateful path based parallel fuzzing for protocols in autonomous vehicles,” in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, ser. DAC ’24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3649329.3658270

[28] F. Ma, Y. Chen, M. Ren, Y. Zhou, Y. Jiang, T. Chen, H. Li, and J. Sun, “Loki: State-aware fuzzing framework for the implementation of blockchain consensus protocols.” in *NDSS*, 2023.

[29] Y. Chen, F. Ma, Y. Zhou, Y. Jiang, T. Chen, and J. Sun, “Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model,” in *2023 IEEE Symposium on Security and Privacy (SP).* IEEE, 2023, pp. 2517–2532.

[30] BlackDuck, “Defensics fuzz testing,” https://www.blackduck.com/fuzz-testing.html, 2024, accessed at November 6, 2024.

[31] jtpereyda, “boofuzz: Network protocol fuzzing for humans,” https://github.com/jtpereyda/boofuzz, 2025, accessed at May 11, 2025.

[32] V.-T. Pham, M. Böhme, and A. Roychoudhury, “Aflnet: a greybox fuzzer for network protocols,” in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST).* IEEE, 2020, pp. 460–465.

[33] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, “Polar: Function code aware fuzz testing of ics protocol,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[34] Google, “american fuzzy lop,” https://github.com/google/AFL, 2024, accessed at October 11, 2024.

[35] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury, “Large language model guided protocol fuzzing,” in *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*, 2024.

[36] J. Godoy, J. P. Galeotti, D. Garbervetsky, and S. Uchitel, “Enabledness-based testing of object protocols,” *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3415153

[37] V. Garousi, A. B. Keleş, Y. Balaman, Z. Özdemir Güler, and A. Arcuri, “Model-based testing in practice: An experience report from the web applications domain,” *Journal of Systems and Software*, vol. 180, p. 111032, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001291

[38] S. Prashast, “Firmfuzz: Automated iot firmware introspection and analysis./prashast srivastava, hui peng, jiahao li, hamed okhravi, howard shrobe, and mathias payer,” in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things (IoT S&P’19)*, 2019, pp. 15–21.

[39] J. Zaddach, L. Bruno, A. Francillon, D. Balzarotti *et al.*, “Avatar: A framework to support dynamic security analysis of embedded systems’ firmwares.” in *NDSS*, vol. 14, no. 2014, 2014, pp. 1–16.

[40] D. D. Chen, M. Woo, D. Brumley, and M. Egele, “Towards automated dynamic analysis for linux-based embedded firmware.” in *NDSS*, vol. 1. The Internet Society, 2016, pp. 1–1.

[41] X. Wang, Y. Sun, S. Nanda, and X. Wang, “Looking from the mirror: Evaluating {IoT} device security through mobile companion apps,” in *28th USENIX security symposium (USENIX security 19)*, 2019, pp. 1151–1167.

[42] K. Liu, M. Yang, Z. Ling, Y. Zhang, C. Lei, J. Luo, and X. Fu, “Riotfuzzer: Companion app assisted remote fuzzing for detecting vulnerabilities in iot devices,” in *Proceedings of the 31th Conference on Computer and Communications Security (CCS’24)*, 2024.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper presents Camveil, a new fuzzer for security cameras. The core insight is that vulnerabilities can be triggered by the interplay of different protocols such as HTTP, RTSP, and ONVIF, an issue that single-protocol fuzzers may not find. To find these cross-protocol bugs, the authors designed a Camera Status Model to capture key runtime states and guide the generation of test sequences. The framework also includes a dedicated Logic Vulnerability Monitor that detects not only crashes but also domain-specific logical flaws, such as a frozen video stream. The authors evaluated Camveil on nine commercial IP cameras and discovered 22 previously unknown vulnerabilities.

A.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Independent Confirmation of Important Results with Limited Prior Research
- Identifies an Impactful Vulnerability

A.3. Reasons for Acceptance

The paper was accepted due to its strong central idea, novel methodology, and impressive, practical results. It addresses an important and overlooked real-world problem. The discovery of 22 new bugs in nine commercial cameras from six different companies provides compelling evidence of the technique’s effectiveness. A particular strength highlighted by reviewers is the Logic Vulnerability Monitor, which provides a tangible way to find tricky but serious logic bugs like video freezes. The paper is well-written, and the case studies effectively demonstrate the proposed technique’s ability to detect vulnerabilities.

A.4. Noteworthy Concerns

Manual Effort and Scalability: Reviewers identify that the approach requires a degree of manual effort, particularly in building the initial packet corpus, labeling packets, and verifying candidate vulnerabilities.