Through the Authentication Maze: Detecting Authentication Bypass Vulnerabilities in Firmware Binaries

Nanyu Zhong[1,2,3,4], Yuekang Li[5], Yanyan Zou[1,2,3,4,†], Jiaxu Zhao[1,2,3,4], Jinwei Dong[1,2,3,4], Yang Xiao[1,2,3,4], Bingwei Peng[1,2,3,4], Yeting Li[1,2,3,4], Wei Wang[1,2,3,4], Wei Huo[1,2,3,4,†]
[1]Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, China
[4]Beijing Key Laboratory of Network Security and Protection Technology, China
[5]University of New South Wales, Australia
{zhongnanyu, zouyanyan, zhaojiaxu, dongjinwei, xiaoyang, pengbingwei, liyeting, wwei, huowei}@iie.ac.cn, yuekang.li@unsw.edu.au

***Abstract*—Embedded web services are widely integrated into network devices such as routers and gateways. These services are often exposed to public networks, making them attractive targets for authentication bypass attacks. Such vulnerabilities allow attackers to gain privileged access without valid credentials, posing serious risks to device integrity and network security. Existing detection techniques rely heavily on manual analysis or rigid heuristics, making them ineffective against diverse and evolving authentication schemes. We present *AuthSpark*, a novel dynamic analysis framework for detecting authentication bypass vulnerabilities in firmware binaries. *AuthSpark* leverages execution trace similarity between successful and failed authentication attempts to locate credential checks. It then tracks authentication-related variable propagation to identify authentication success logic. Finally, it employs a customized greybox fuzzer with task-specific power scheduling and mutation strategies to explore bypass paths. We evaluate *AuthSpark* on firmware from 32 real-world devices containing 14 known vulnerabilities. *AuthSpark* successfully identifies 42 out of 44 credential checks and detects 14 of the known vulnerabilities. More importantly, when applied to the latest firmware versions, *AuthSpark* discovers six zero-day authentication bypass vulnerabilities, four of which received official assignments (three CVEs and one PSV). These results highlight *AuthSpark*'s effectiveness and its potential to uncover critical security flaws in real-world systems.**

I. Introduction

Embedded systems [43] are key components of modern network infrastructure, including routers, firewalls, gateways. Their security is vital to protect users and enterprises globally. Web services are widely deployed on these devices and often exposed to public networks for remote access. This exposure creates a large and vulnerable attack surface.

In recent years, many techniques have been proposed to detect vulnerabilities in web services of embedded system firmware [5], [29], [49], [51]. Most approaches focus on memory-related flaws, while logic vulnerabilities remain understudied. A key type of such logic vulnerability is authentication bypass. These vulnerabilities allow attackers to gain privileged access without valid credentials. Once bypassed, attackers can perform post-authentication operations, modify configurations, and compromise connected networks. Such flaws are especially dangerous, as they may grant access needed to exploit other vulnerabilities.

To detect authentication bypass vulnerabilities, researchers have proposed several techniques, most of which are over a decade old. Two representative examples are Firmalice [38] (2015) and Weasel [37] (2013). Both adopt a two-phase strategy: first, they locate authentication-related code either manually or heuristically; then, they apply symbolic execution to explore paths from unauthenticated inputs that reach privileged operations without proper checks. Later, general-purpose frameworks such as Avatar [48] and Symbion [17] were introduced to enhance symbolic execution for firmware analysis, both of which can be leveraged by authentication bypass vulnerability detection.

A key limitation of existing approaches is their reliance on heuristics or manual effort to identify authentication-related code. This restricts them to specific authentication patterns or web requests. However, web services often use diverse and evolving authentication schemes. Existing tools lack the generalizability to handle such diversity. As a result, new vulnerabilities continue to emerge. For example, CVE-2023-20198 [7], an authentication bypass flaw in Cisco IOS XE, compromised over 140,000 systems and was actively exploited in the wild [6], [8], [34]. To address this, we need to develop metaheuristics that are general enough to support diverse authentication mechanisms.

By analyzing authentication bypass vulnerabilities reported in recent years, we observe a common underlying rationale. Despite diverse authentication mechanisms, a vulnerability ex-

†These authors are co-corresponding authors.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.242757
www.ndss-symposium.org

ists if there is an execution path that reaches the authentication success state without proper credential checks. Based on this insight, detecting such vulnerabilities requires identifying three key elements: ❶ the credential checks; ❷ the code representing the authentication success state; ❸ the bypass paths between them. Advances in symbolic execution and fuzzing have made exploring bypassing paths relatively straightforward. Thus, the main challenge lies in accurately identifying credential checks and authentication success codes. While existing techniques partially incorporate this idea, they do not focus explicitly on solving this problem. Therefore, we propose to focus on addressing this challenge systematically and effectively.

We make two key observations to precisely identify credential checks and authentication success codes. First, across different authentication mechanisms, execution traces for successful and failed authentication requests are highly similar before the credential checks but diverge significantly afterward. This change in similarity can signal the location of the credential checks. Second, the results of credential checks propagate through specific variables. Tracking the propagation of these variables helps locate the code corresponding to authentication success.

Based on these observations, we propose *AuthSpark*, a metaheuristic-based approach for detecting authentication bypass vulnerabilities in firmware binaries. First, *AuthSpark* uses dynamic analysis to generate execution traces for successful and failed authentication requests. It then compares these traces to identify credential checking code, referred to as *credential verification statements* in our methodology. Next, *AuthSpark* tracks the propagation of the variables storing the results of credential checks to locate authentication success codes, termed *authentication-success basic blocks*. Finally, *AuthSpark* employs a specialized greybox fuzzer to explore authentication bypass paths, using power scheduling and mutation strategies tailored to this task.

We conducted a comprehensive evaluation of *AuthSpark* on a dataset of 32 devices containing 14 known vulnerabilities. For localization of credential verification statements, *AuthSpark* correctly identified 42 out of 44 locations. For vulnerability detection, it successfully detected 14 known vulnerabilities using the identified authentication-success basic blocks, demonstrating the effectiveness of our approach. More importantly, when applied to the latest firmware of the 32 devices, *AuthSpark* discovered six zero-day vulnerabilities, four of which received official assignments (three CVEs and one PSV).

In summary, we make the following contributions:

- We analyze the rationale behind authentication bypass vulnerabilities and reveal the limitations of existing methods and metaheuristics in detecting them.
- We propose the first dynamic approach for detecting authentication bypass vulnerabilities. It leverages dynamic information to identify credential verification statements and authentication-success basic blocks, and uses this information to help a customized fuzzer effectively detect authentication bypass vulnerabilities.
- We implement *AuthSpark*, a novel dynamic analysis framework that automatically dissects authentication logic, accurately detects known authentication bypass vulnerabilities, and uncovers six previously unknown ones.

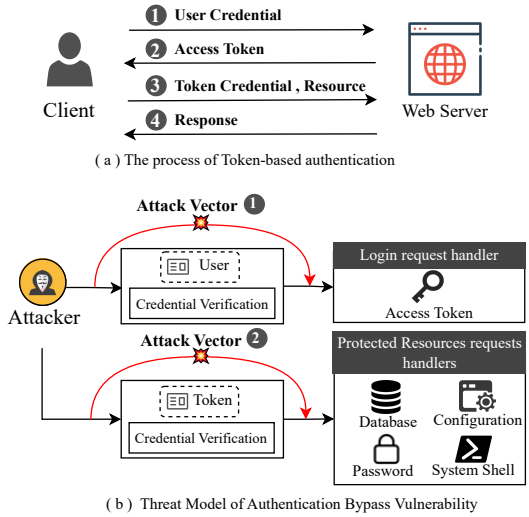

Fig. 1: Authentication Process and Vulnerability Illustration

II. Background and Related Work

A. Authentication Module

Authentication is the process by which your identity is confirmed through the use of some kind of credential [15]. As illustrated in Figure 1(a), a typical authentication workflow for web services consists of four steps: ❶ the client submits credentials to the web server, ❷ the web server validates these credentials and issues an access token upon successful verification, ❸ the user presents this token to access protected resources, and ❹ the server verifies the token before granting access to the requested resources. This token-based authentication pattern has become the de facto standard in the web services [20] of modern embedded systems.

Modern embedded systems support multiple authentication mechanisms to accommodate diverse use cases and security requirements, each falling into one of two fundamental categories: (1) User credentials: authentication factors used to prove user identity during the login process. These include passwords, PIN codes, and digital certificates. (2) Token credentials: authentication proofs issued by the server after successful login, used to maintain authenticated sessions and access protected resources. These include session cookies, JWT tokens, OAuth access tokens, API keys, and session IDs.

B. Related Work

While various studies have examined authentication bypasses targeting specific vulnerability root causes or particular

authentication mechanisms, such as misconfigurations [36], hardcoded credentials [19], [24], or specific authentication mechanisms [2], [12], [16], [27], [42], we focus on general-purpose approaches that can systematically identify authentication bypass vulnerabilities across different authentication mechanisms.

Firmalice [38] and Weasel [37] represent general-purpose detection approaches for authentication bypass vulnerabilities, not targeting specific authentication mechanisms. Both follow a two-phase process: first locating authentication-related code either heuristically or manually, then employing symbolic execution to discover paths from unauthenticated user inputs that bypass authentication checks and reach authentication success code. Firmalice [38] requires firmware security policies, manually provided by human analysts, to locate privileged codes that should only execute after successful authentication. Once located, it symbolically finds paths from user inputs requiring unique constraint solutions to indicate bypasses. Unlike Firmalice, Weasel [37] focuses on backdoor detection by finding any feasible path from user input to backdoor functions that avoids authentication validation code. To identify authentication validation code, Weasel employs a dynamic approach: it first collects execution traces, then it tries to construct decision trees through differential analysis and identifies the dominator nodes as the authentication validation codes. As a result, the limited execution traces collected significantly constrains Weasel's identification ability.

Despite enhancements from tools like Avatar [48] and Symbion [17] for binary symbolic execution, these methods share critical limitations: they rely on manual annotation or heuristic-based identification of unauthenticated user inputs and authentication success code, failing to provide a comprehensive, generalizable solution for identifying these elements.

In the domain of fuzzing for embedded systems, various works have improved different aspects of the fuzzing process. FirmFuzz [40], FirmAFL [53] and greenhouse [41] focus on emulation adaptation, while IoTFuzzer [4], SRFuzzer [49], Snipuzz [14] and housefuzz [47] enhance mutation strategies and feedback mechanisms. However, none specifically target authentication bypass vulnerabilities.

III. Threat Model

This paper focuses on attacks targeting web services in embedded systems, specifically investigating logic flaws that enable anonymous users to bypass authentication mechanisms. We define authentication bypass vulnerabilities as security flaws allowing attackers to circumvent authentication mechanisms, directly executing privileged functionality or accessing protected resources without proper credentials. From a security perspective, web service endpoints are either protected (requiring valid token credentials) or public (e.g., login operations, static resources). Based on this classification, authentication bypass vulnerabilities can only occur at endpoints involving credential verification: login operations and token-protected resources. As illustrated in Figure 1(b), we identify two primary attack vectors: ❶ bypassing user credential verification to obtain valid token credentials without providing correct user credentials; ❷ bypassing token credential verification to directly access protected resources without valid token credentials. We assume attackers can craft arbitrary HTTP requests and have knowledge of API endpoints but lack valid user or token credentials. This threat model guides our analysis to identify authentication related codes and detect potential bypass paths in real-world embedded systems.

IV. Motivating Example

A. The Authentication Bypass Vulnerability

Figure 2 presents an authentication bypass vulnerability discovered by our tool in a real-world embedded systems. The device implements a web service based on the lighttpd framework [28] with a custom authentication module. All HTTP requests are processed through function sub_4040, which determines authentication outcomes by invoking either ntlm_authentication or basic_authentication for NTLM [10] or Basic [35] authentication respectively. In addition, this backend has configuration file which determines requests from which type of user agents should be handled by NTLM authentication.

Normally, Basic authentication and NTLM authentication are two separate routines. For basic authentication, usernames and passwords are used. The front-end stores the username and password as a base64 string in the Authorization entry of the request header sent to the back-end. The workflow of a successful Basic authentication is shown by steps ❶ to ❻ in Figure 2. For NTLM authentication, the back-end first checks the name of the user agent (step ❶), then invokes parse_ntlm_info (step ❷) to parse the NTLM payload and set the smb_info.qflag to 1 if the payload is successfully parsed. After that, the back-end invokes ntlm_authentication (step ❹), checks the authentication information (step ❺), and goes through another check (step ❻) before success (step ❼).

However, an attacker can craft a malicious request to bypass the authentication using the User–Agent entry of an NTLM authentication request and the Authorization entry of a Basic authentication request. The main reason is that the mixture of entries from both types of authentication triggers a corner case where both authentication routines think it is the responsibility of the other routine to handle the request and therefore no authentication is done in the end. First, since User–Agent is WebDrive, the back-end sets the ua_flag to 3 (step ❶) and will invoke ntlm_authentication later on (step ❺). Second, because the Authorization entry is for Basic authentication, parse_ntlm_info will fail and the value of smb_info.qflag will remain 2. Therefore, in ntlm_authentication, the authentication check for the Authorization entry will be skipped (steps ❻,❼,❽). And ntlm_authentication returns authentication success without doing any authentication checks at all (step ❾).

B. Limitations of Existing Techniques

From the vulnerability introduced in Figure 2, we can observe that different from memory-related vulnerabilities, authentication bypass vulnerabilities are logic vulnerabilities

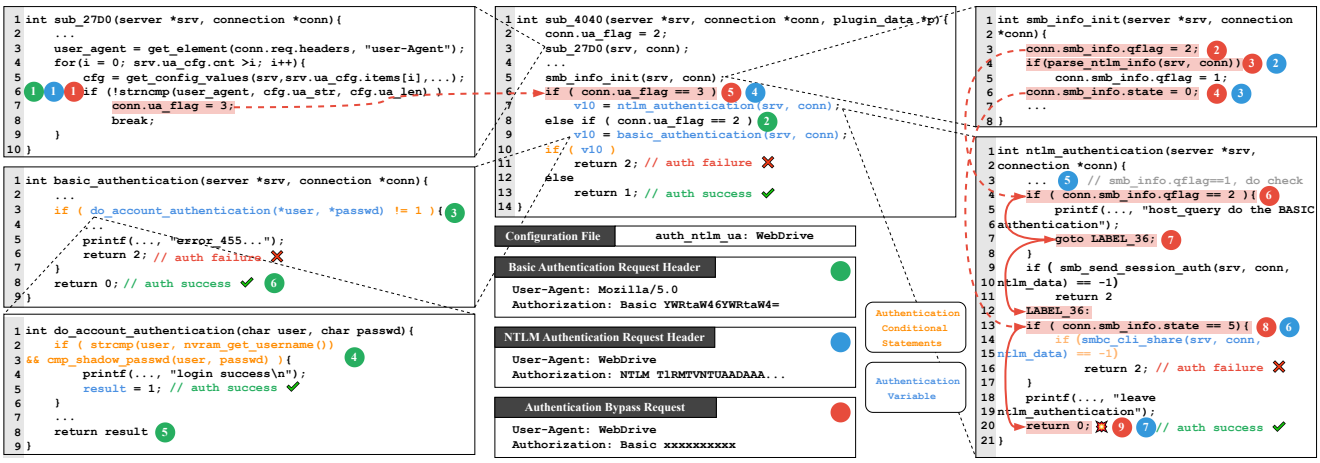

Fig. 2: Overview of CVE-2025-2492.

which do not emit explicitly observable signals (like crashes) when triggered. Therefore, it is important to make the triggering of such vulnerabilities observable to detect them. For this purpose, understanding the authentication workflows is necessary, which is the focus of existing techniques.

Most existing techniques focus on leveraging particular authentication patterns or specific web requests. However, different targets can have very different authentication mechanisms and processes. Moreover, even one target can have multiple authentication routines, such as Basic authentication and NTLM authentication in the motivating example. Therefore, existing techniques are often specific to certain types of authentication bypass patterns, lacking generalizability and scalability.

As shown in Figure 2, the statement highlighted at lines 4 and 5 of do_account_authentication is recognized as the authentication-related codes by Firmalice because it contains "login success", however, NTLM-based routines lack such lexical clues, causing Firmalice to miss the NTLM-related vulnerabilities entirely. For Weasel, a representative heuristic-based approach, it can not handle such multi-authentication mechanisms either. It can only identify sub_4040 as the authentication validation function, and misses two other authentication validation functions (i.e., basic_authentication and ntlm_authentication).

C. Observations and Inspirations

To find a general solution for detecting authentication bypass vulnerabilities regardless of the authentication workflows, we need to understand their nature. The rationale of authentication bypass vulnerabilities is simple: there is an execution path to reach the code representing the authentication success states without going through proper credential checks. Accordingly, we can simplify the detection of authentication bypass vulnerabilities into three key steps: ❶ Identifying the credential check results; ❷ Identifying the authentication success codes; ❸ Explore the bypass paths. Therefore, in this work, we propose to focus effort on solving these key issues, especially the precise identification of the authentication success codes and credential check results.

We introduce the key concepts to describe these types of authentication related code. *Authentication variables* (AVs) are the variables that store the authentication outcomes. Examples of authentication variables are highlighted in blue in Figure 2. *Authentication conditional statements* are conditional statements that check the values of authentication variables and diverge the control flow. Specifically, for each authentication mechanism, there is a unique authentication conditional statement called *credential verification statement* (CVS), and it is usually the initial authentication conditional statement for an authentication mechanism. Its functionality is to verify the correctness of the credential provided in the request. Examples of authentication conditional statements are highlighted in orange in Figure 2. There are two types of basic blocks whose executions depend on the authentication conditional statements. *Authentication-success basic blocks* are basic blocks that are executed if and only if authentication succeeds, while *authentication-failure basic blocks* are executed otherwise.

We make two observations about the characteristics of authentication related code:

- **Observation 1**. For authentication success requests and authentication failure requests, their execution traces are highly similar before the CVSs but are very different after the CVSs. For example, in Figure 2, no matter whether a request causes authentication to succeed or fail, the program always needs to set certain flags (sub_27D0, smb_info_init). However, after reaching the CVSs, the execution traces are very different with failures terminate earlier.
- **Observation 2**. Identifying CVSs can help improve the identification of authentication-success basic blocks. The authentication-success basic blocks are dominated by the basic blocks containing authentication conditional statements that in turn can be determined by CVSs. In other words, all execution paths reaching the authentication-success basic

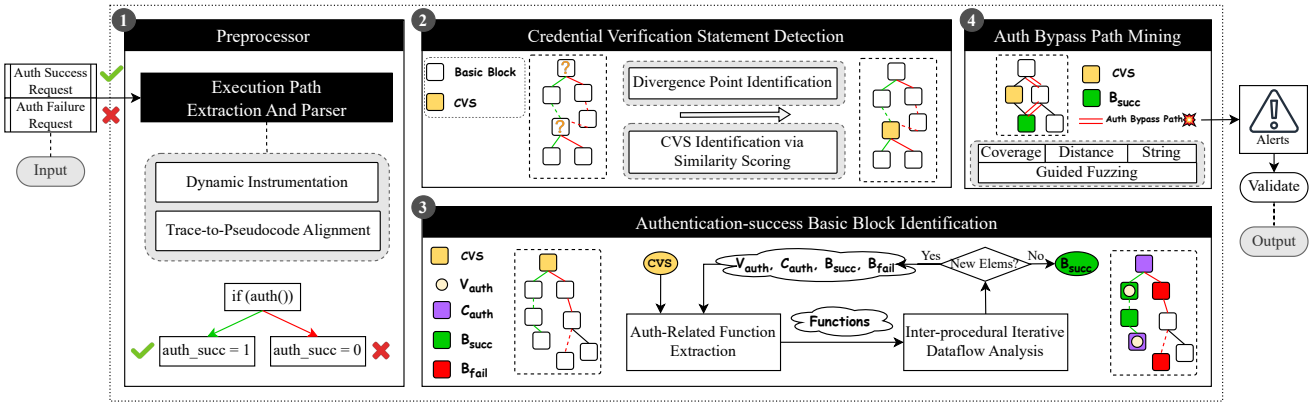

Fig. 3: Overview of *AuthSpark*

blocks must also pass through the authentication conditional statements.

Based on observations, we can get inspiration for designing the techniques used by *AuthSpark* to identify precisely the CVSs and authentication-success basic blocks.

- **Inspiration 1**. We can craft authentication success requests and authentication failure requests and differentiate their execution traces to locate CVSs. The execution traces have high similarity before the CVSs but diverge sharply afterwards, resulting in low similarity.
- **Inspiration 2**. The authentication variables used by CVSs are propagated to affect the execution of authentication-success basic blocks in two ways: (1) they are returned upward to calling functions for further validation, and (2) they flow forward to subsequent program points where they control access to resources and privileged operations. The propagation patterns are easier to identify and they can help with the identification of authentication-success basic blocks.

V. Methodology

Figure 3 depicts the overview of *AuthSpark*. It works mainly in four phases. ❶ *Preprocessing.* *AuthSpark* collects pairs of dynamic execution traces by dynamic instrumentation, each of which comprises both successful and failure authentication requests, and transforms the traces into sequences of pseudocode statements for subsequent analysis. ❷ *Credential Verification Statement Detection.* For each of the trace pairs, *AuthSpark* extracts all divergence points using differential analysis, and identifies the CVS from these candidates based on the ranking derived from their trace-similarity scores. ❸ *Authentication-success Basic Block Identification.* Starting from the detected CVS, *AuthSpark* extracts authentication-related functions and performs iterative inter-procedural dataflow analysis to identify authentication variables and their propagation paths. Through this analysis, it locates all authentication conditional statements and determines the authentication-success basic blocks related to these statements' outcomes. ❹ *Authentication Bypass Path Mining.* Three complementary fuzzing strategies, i.e., coverage-guided, distance-guided, and string-guided fuzzing, are employed to explore paths from unauthenticated inputs toward authentication-success basic blocks. When any of these blocks is reached, a validation check is performed to verify whether an authentication bypass vulnerability is discovered.

A. Preprocessing

Paired Authentication Requests. To identify authentication related code, *AuthSpark* performs differential analysis on a pair of authentication requests: one valid (success) and one invalid (failure). This minimal input requirement applies to both user and token credentials, as defined in Section II-A. For example, in token-based authentication, the request pair is generated by capturing a successful request (Request A) and mutating only the token field to create an invalid variant (Request B), such as replacing the token with a dummy value like "AAA". Such a scoped mutation ensures that any behavioral divergence in execution traces arises from the credential verification, which is essential for subsequent analysis.

Dynamic Instrumentation. With the request pair constructed, *AuthSpark* uses QEMU-based dynamic instrumentation to record execution traces for both requests. The instrumentation logs basic block entries to produce a precise control-flow trace. These traces provide the foundation for identifying divergences in execution paths attributable to credential checks.

Trace-to-Pseudocode Alignment. To enable subsequent variable- and statement-level analysis, which is difficult to perform directly on assembly, *AuthSpark* aligns each dynamic trace with its corresponding pseudocode representation generated by IDA Pro [21]. *AuthSpark* constructs an interprocedural control flow graph (ICFG) at the pseudocode level. Each ICFG node corresponds to a decompiled basic block containing high-level statements. To reconstruct semantically accurate execution paths, each basic block in the dynamic trace is mapped to its corresponding pseudocode block by resolving its constituent instructions. To handle interprocedural and concurrent behaviors (e.g., thread interleaving), *AuthSpark*

recovers and propagates call-stack context at each call site. This produces an ordered, context-aware pseudocode trace faithfully representing the observed execution.

To support identification of authentication-success basic blocks, *AuthSpark* further constructs statement-level control dependence graph (CDG) and data dependence graph (DDG) for each function in the binary.

B. Credential Verification Statement Detection

The *credential verification statement* (CVS) refers to the code location where authentication outcomes are determined, typically implemented as conditional branches that validate user or token credentials (e.g., the result of strcmp). Based on **Inspiration 1**, *AuthSpark* employs a two-stage differential analysis approach: first identifying divergence points from execution trace pairs, then scoring them to pinpoint the most probable CVS.

Divergence Point Identification. To identify all divergence points between two execution traces, we employ the Longest Common Subsequence (LCS) algorithm [3]. We treat the last element of this identified common subsequence as a candidate divergence point, since it marks the final shared behavior before the corresponding divergent segment. However, directly applying LCS to raw traces, which often contain tens of thousands of basic blocks, is computationally prohibitive [45]. To address this, we adopt a divide-and-conquer strategy based on call stack contexts. We first abstract each trace into a sequence of call-stack entries at function call sites, where each entry is represented as a "(call-stack, trace index)" tuple. Applying LCS to these call-stack sequences yields reliable alignment positions (detailed in Appendix A), which partition the original traces into smaller aligned segments. Within each segment, LCS can then be applied efficiently for fine-grained divergence detection, and the last element of each local LCS result is extracted as a candidate divergence point.

CVS Identification via Similarity Scoring. Not all divergence points correspond to credential checks. Guided by **Inspiration 1**, which characterizes the CVS as the boundary between identical and diverging execution, we assign a similarity-based score to each divergence point. To operationalize this, we use the RO algorithm [44], which computes a normalized similarity score in the [0, 1] by identifying matching blocks while preserving their relative order.

The score for each divergence point d is defined as:

$$P(d) = Sim_{\text{before}}^{\alpha} \times (1 - \beta \times Sim_{\text{after}}) \tag{1}$$

where Sim_{before} and Sim_{after} denote the sequence similarity before and after d, respectively. The exponent α (with $\alpha \geq 1$) amplifies the reward for near-identical prefixes, while β (restricted to $0 < \beta \leq 1$) applies a moderate penalty to post-divergence similarity. *AuthSpark* uses Equation 1 to assign each divergence point a score and produce a prioritized ranking of candidate CVSs. Analysts can then confirm the true CVS by examining candidates in order of priority, as higher-ranked divergence points, particularly those appearing earlier in the execution traces, are more likely to correspond to credential checks. In practice, the configuration $(\alpha, \beta) = (3, 0.2)$ provides a reliable ranking across devices, and our evaluation (Section VII-B) shows that the top-ranked candidate is the correct CVS for most firmware samples. As shown in Figure 2, this method successfully identifies the credential statement (Label ❹) in do_account_authentication.

C. Authentication-success Basic Block Identification

Starting from the identified CVS, *AuthSpark* first extracts the initial authentication variables. It then models the subsequent analysis as a standard forward data-flow problem on the ICFG and performs an iterative data-flow analysis to derive new authentication variables. Based on **Inspiration 2**, *AuthSpark* tracks how these variables propagate and are used in the program so that it can systematically identify the authentication conditional statements and the authentication-success basic blocks that depend on the CVS. The overall workflow appears in Alg. 1.

During the iterative analysis, *AuthSpark* maintains three types of authentication-related code elements, each represented as a set:

1) Authentication variables (AVs), denoted by $\mathcal{V}_{\text{auth}}$. Each element has the form (v, val, x), where v is a variable, val is its value, and x is the authentication success or failure outcome carried by v.

2) Authentication conditional statements (ACSs), denoted by $\mathcal{C}_{\text{auth}}$. Each element has the form $(c, branch, x)$, where c is a conditional statement, $branch$ denotes the condition is true or not, and x is the authentication outcome.

3) Authentication-success and authentication-failure basic blocks (ASBBs, AFBBs), denoted by $\mathcal{B}_{\text{succ}}$, $\mathcal{B}_{\text{fail}}$, respectively. These blocks appear on success or failure traces of the authentication logic.

As detailed in Alg. 1, *AuthSpark* first inserts the CVS into $\mathcal{C}_{\text{auth}}$ as the starting point of the authentication logic (line 2–3). The *getBranches* function extracts the branch outcomes of statement s from the successful and failed authentication traces. It then applies *MapDynamicInfoToICFG* to map the dynamic execution information from authentication request pair traces onto ICFG nodes. This information includes function return values, variable values, and function argument values, and this mapping supports precise analysis in later stages. The analysis proceeds iteratively to identify ASBBs. In each iteration, it selects functions containing any authentication-related code elements (line 7, *GetAuthRlateFunc*) and analyzes their basic blocks in topological order (line 11). For each basic block, *AuthSpark* applies the customized *AuthGen* and *AuthKill* functions (lines 12–16) to compute the generation and kill sets, which are designed to track AVs. Based on these sets, *AuthSpark* updates the OUT set of the block and extends $\mathcal{V}_{\text{auth}}$. It then calls *UpdateAuthConds* and *UpdateAuthBlocks* (lines 17–18) to update $\mathcal{C}_{\text{auth}}$, $\mathcal{B}_{\text{succ}}$, and $\mathcal{B}_{\text{fail}}$. The process continues until no new authentication-related code elements appear, (i.e., $\mathcal{X} = \mathcal{X}^{\text{prev}}$), at which point the analysis converges and produces the final ASBBs.

Alg. 1: Auth-success basic blocks Identification

Input : Credential verification statement s, program's ICFG execution traces p_{succ} and p_{fail}
Output: Authentication-success basic blocks $\mathcal{B}_{\text{succ}}$

1 $\mathcal{V}_{\text{auth}} \leftarrow \emptyset$, $\mathcal{B}_{\text{succ}} \leftarrow \emptyset$, $\mathcal{B}_{\text{fail}} \leftarrow \emptyset$ $\mathcal{X}^{\text{prev}} \leftarrow \emptyset$;
2 $(Br_{\text{succ}}, Br_{\text{fail}}) \leftarrow$ *getBranches*$(s, (p_{\text{succ}}, p_{\text{fail}}))$;
3 $\mathcal{C}_{\text{auth}} \leftarrow \{(s, Br_{\text{succ}}, \text{succ}), (s, Br_{\text{fail}}, \text{fail})\}$;
4 *MapDynamicInfoToICFG*(p_{succ}, p_{fail}, ICFG);
5 **while** $\exists\, \mathcal{X} \in \{\mathcal{V}_{auth}, \mathcal{C}_{auth}, \mathcal{B}_{succ}, \mathcal{B}_{fail}\} : \mathcal{X} \neq \mathcal{X}^{prev}$ **do**
6 $\mathcal{X}^{\text{prev}} \leftarrow \{\mathcal{V}_{\text{auth}}, \mathcal{C}_{\text{auth}}, \mathcal{B}_{\text{succ}}, \mathcal{B}_{\text{fail}}\}$;
7 relevantFuncList = *GetAuthRlateFunc*($\mathcal{C}_{\text{auth}}$, $\mathcal{V}_{\text{auth}}$, ICFG);
8 **foreach** $f \in$ *relevantFuncList* **do**
9 **foreach** *block* $B \in f$*.blocks* **do**
10 OUT$[B] \leftarrow \emptyset$;
11 **foreach** *block* $B \in f$*.blocks in topological order* **do**
12 IN$[B] \leftarrow \bigcup_{P \in \text{pred}(B)}$ OUT$[P]$;
13 $\text{gen}_B \leftarrow$ *AuthGen*$(B, \mathcal{V}_{\text{auth}}, \mathcal{C}_{\text{auth}})$;
14 $\text{kill}_B \leftarrow$ *AuthKill*$(B, \mathcal{V}_{\text{auth}})$;
15 OUT$[B] \leftarrow \text{gen}_B \cup (\text{IN}[B] - \text{kill}_B)$;
16 $\mathcal{V}_{\text{auth}} \leftarrow \mathcal{V}_{\text{auth}} \cup$ OUT$[B]$;
17 $\mathcal{C}_{\text{auth}} \leftarrow$ *UpdateAuthConds*$(B, \mathcal{V}_{\text{auth}}, \mathcal{C}_{\text{auth}}, \mathcal{B}_{\text{succ}}, \mathcal{B}_{\text{fail}}, p_{\text{succ}}, p_{\text{fail}})$;
18 $(\mathcal{B}_{\text{succ}}, \mathcal{B}_{\text{fail}}) \leftarrow$ *UpdateAuthBlocks*$(B, \mathcal{C}_{\text{auth}}, \mathcal{B}_{\text{succ}}, \mathcal{B}_{\text{fail}})$;
19 **return** $\mathcal{B}_{succ}$;

Alg. 2: Authentication Variables Gen/Kill Analysis

1 **Func** `AuthGen(`B, $\mathcal{V}_{auth}$, $\mathcal{C}_{auth}$`)`
2 $\text{gen}_B \leftarrow \emptyset$;
3 **foreach** *statement* $s \in B$ **do**
4 **if** *s is assign v = val and ControlDepsPost(s)* $\subseteq \mathcal{C}_{auth}$ **then**
5 x $\leftarrow$ *determineAuthOutcome*(s, $\mathcal{C}_{auth}$);
6 $\text{gen}_B \leftarrow \text{gen}_B \cup \{(\text{v, val, x})\}$;
7 **if** *s is copy assign v = v' and* $\exists$ *(v', val', x)* $\in \mathcal{V}_{auth}$ **then**
8 $\text{gen}_B \leftarrow \text{gen}_B \cup \{(\text{v, val', x})\}$;
9 **if** *s is return stmts with* v_{ret} *and (* $v_{ret} \in vars(\mathcal{V}_{auth})$ *or ControlDepsPost(s)* $\cap \mathcal{C}_{auth} \neq \emptyset$*)* **then**
10 func $\leftarrow$ *getContainingFunc*(s);
11 $\text{val}_{succ}, \text{val}_{fail} \leftarrow$ *evaluateRetVal*(s, $\mathcal{V}_{auth}$);
12 **if** $val_{succ} \neq val_{fail}$ **then**
13 $\text{gen}_B \leftarrow \text{gen}_B \cup \{(\text{func}, \text{val}_{succ}, \text{succ}), (\text{func}, \text{val}_{fail}, \text{fail})\}$;
14 **return** gen_B;
15 **Func** `AuthKill(`B, $\mathcal{V}_{auth}$`)`
16 $\text{kill}_B \leftarrow \emptyset$;
17 **foreach** *statement* $s \in B$ **do**
18 **if** *s redefines v and* $\exists(v, \_, \_) \in \mathcal{V}_{auth}$ **then**
19 $\text{kill}_B \leftarrow \text{kill}_B \cup \{(\text{v}, \_, \_) \in \mathcal{V}_{\text{auth}}\}$;
20 **return** $kill_B$;

AuthSpark maintains the authentication variable set $\mathcal{V}_{\text{auth}}$ using customized *AuthGen* and *AuthKill* functions (Alg. 2). These two functions compute the generation and kill sets for AVs during data-flow analysis, supporting the analysis of local variables, global variables, and function return variables. Specifically, the *AuthGen* function identifies AVs based on three conditions. A variable is marked as an authentication variable if it satisfies any of the following conditions:

1) Control dependency condition (lines 4-6). If an assignment's control dependencies come from a strict subset of the identified authentication conditional branches in $\mathcal{C}_{\text{auth}}$, and all these branches correspond to the same authentication outcome (either success or fail), the variable is marked as an AV. *AuthSpark* then records the associated authentication outcome for this assignment via *determineAuthOutcome*. The strict subset requirement ensures that the assignment occurs exclusively under a single authentication outcome, thereby excluding variables whose control dependencies span mixed authentication outcomes. *AuthSpark* leverages the constructed pseudocode-level CDG to obtain the control dependencies for each assignment (*ControlDepsPost* in Alg. 2). The "Post" notation indicates we only consider control dependencies occurring after the CVS, as we focus on how authentication results influence subsequent behavior. For example, as shown in Figure 4, *AuthSpark* only considers control dependencies starting from "C1".

2) Copy propagation condition (lines 7-8). If a variable receives its value from an authentication variable in $\mathcal{V}_{\text{auth}}$ through a copy statement, it inherits the authentication status.

3) Function return variable condition (lines 9-13). If a return statement contains an authentication variable or its control dependencies include any ACS, *AuthSpark* first retrieves its enclosing function via *getContainingFunc* and evaluates the return values under success and failure contexts using *evaluateRetVal*. If the two return values differ, the function's return value is also considered an authentication variable.

Correspondingly, the *AuthKill* function (lines 15–20) removes authentication variables that are redefined, ensuring the correct maintenance of authentication variable states. In this algorithm, (v,_,_) denotes all AV elements whose variable component is v. It is important to note that killing a variable only removes its authentication property at the current program point, not at previous points.

Building upon the identified authentication variables, *AuthSpark* needs to further determine which conditional statements and basic blocks are related to authentication logic (Alg. 3). We design the *UpdateAuthConds* and *UpdateAuthBlocks* functions to identify ACSs and ASBBs during iteration.

The *UpdateAuthConds* function employs two strategies to identify ACS: ❶ *Variable-based identification* (lines 3-6). It checks whether a conditional statement uses only authentication variable in $\mathcal{V}_{\text{auth}}$ (extracted via *refVars*). If so, *AuthSpark* uses an SMT solver [46], based on the values of these authentication variables, to infer the authentication outcome of the branch. After the outcome is inferred, *AuthSpark* records the conditional statement together with the evaluated branch direction and its corresponding authentication outcome into $\mathcal{C}_{\text{auth}}$. ❷ *Dynamic trace-based identification* (lines 7-12).

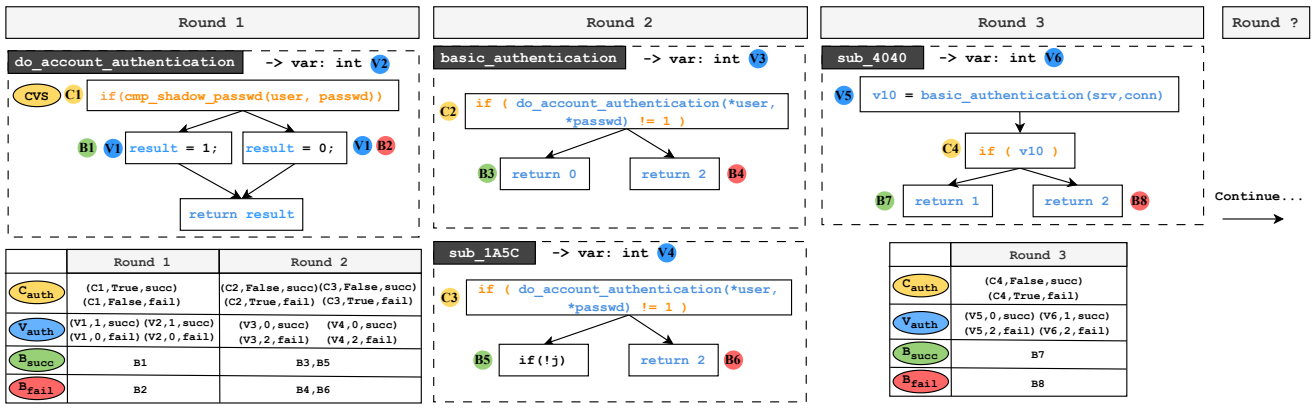

Fig. 4: The ASBBs identification process in the motivation example. "C1–C4" denotes ACSs, "V1–V6" denotes AVs, and "B1–B8" denotes ASBBs and AFBBs. Among them, V2, V3, V4, and V6 are function return variables identified as AVs. The table shows the newly identified authentication-related code elements in each iteration round.

Alg. 3: Authentication Conditionals and Blocks Update

Func UpdateAuthConds ($B, \mathcal{V}_{\text{auth}}, \mathcal{C}_{\text{auth}}, \mathcal{B}_{\text{succ}}, \mathcal{B}_{\text{fail}}, p_{\text{succ}}, p_{\text{fail}}$)
foreach $s \in B$ *and* s *is cond stmts* **do**
  if $refVars(s) \subseteq vars(\mathcal{V}_{auth})$ **then**
    foreach $x \in \{succ, fail\}$ **do**
      branch $\leftarrow$ *EvalBranch*(s, $\mathcal{V}_{auth}$, x);
      $\mathcal{C}_{auth} \leftarrow \mathcal{C}_{auth} \cup \{(\text{s, branch, x})\}$;
  if $s \in$ *dynamic traces* **then**
    foreach $x \in \{succ, fail\}$ **do**
      Br_x = *getBranches*((s, p_x);
      Block_{opp} = *EvalBlock*($\neg\text{Br}_x$, s);
      // $\bar{x}$ is the opposite outcome of x
      if $Block_{opp} \in \mathcal{B}_{\bar{x}}$ **then**
        $\mathcal{C}_{\text{auth}} \leftarrow \mathcal{C}_{\text{auth}} \cup \{(\text{s}, \text{Br}_x, \text{x}), (\text{s}, \neg\text{Br}_x, \bar{\text{x}})\}$;
return $\mathcal{C}_{auth}$;
Func UpdateAuthBlocks(B, $\mathcal{C}_{auth}, \mathcal{B}_{succ}, \mathcal{B}_{fail}$)
s = *FirstStmt*(B);
if *ControlDepsPost(s)* $\subseteq \mathcal{C}_{auth}$ **then**
  x $\leftarrow$ *determineAuthOutcome*(s, $\mathcal{C}_{auth}$);
  $B_x \leftarrow B_x \cup \{B\}$;
return (B_{succ}, B_{fail});

This rule leverages dynamic traces to infer implicit ACSs that are not directly tied to authentication variables. If, in a successful authentication trace, the branch not taken by a conditional statement leads to a basic block in the AFBBs set $\mathcal{B}_{\text{fail}}$, *AuthSpark* infers that taking the opposite branch would result in authentication failure. This behavior is fully consistent with the semantics of ACSs, in which the two branch directions correspond to the two authentication outcomes. Symmetrically, the same reasoning applies to failure traces. Therefore, *AuthSpark* adds both branch directions and their corresponding authentication outcomes to $\mathcal{C}_{\text{auth}}$.

Based on the updated $\mathcal{C}_{\text{auth}}$, the *UpdateAuthBlocks* function determines whether the current basic block should be included in the authentication-success or authentication-failure block sets. It checks whether the control dependencies of the block's first statement form a non-empty subset of $\mathcal{C}_{\text{auth}}$. If so, *AuthSpark* determines the corresponding authentication outcome based on the matched element's outcome in $\mathcal{C}_{\text{auth}}$ via *determineAuthOutcome* and inserts the block into the appropriate basic-block set ($\mathcal{B}_{\text{succ}}$ or $\mathcal{B}_{\text{fail}}$).

Example of the ASBBs Identification Process. Figure 4 illustrates *AuthSpark*'s iterative analysis process on the basic authentication logic in our motivation example. *AuthSpark* first generates pairs of HTTP requests for Basic authentication under the default configuration and identifies the conditional statement C1 as the CVS by analyzing their execution paths. Before the analysis begins, this CVS is added as the initial element of $\mathcal{C}_{\text{auth}}$, e.g., (C1, True, succ).

Through multiple rounds of analysis, *AuthSpark* progressively uncovers authentication logic elements. ❶ In Round 1, *AuthSpark* analyzes the do_account_authentication function, where the assignment result=1 in basic block $B1$ satisfies the control dependency condition; the variable result (denoted as V1 in Fig. 4) is marked as an authentication variable. The subsequent return result statement marks the do_account_authentication function return variable (as V2 shown in Fig. 4) as an authentication variable, because it meets the Function return variable condition. ❷ In Round 2, the analysis expands to all functions calling do_account_authentication . Conditional statements $C2$ and $C3$ are added to $\mathcal{C}_{\text{auth}}$ as they check authentication variable $V2$ in Fig. 4. Notably, function sub_1A5C, though not on the current execution path, is still analyzed because it contains authentication-related logic and calls do_account_authentication. The return variable of basic_authentication marks $V3$ as an authentication variable since its control dependency includes $C2$, which belongs to $\mathcal{C}_{\text{auth}}$. ❸ In Round 3, the assignment v10 = basic_authentication(srv, conn) in sub_4040 satisfies the copy propagation condition, allowing the variable v10 to inherit the authentication property from the basic_authentication function return variable.

Throughout each iteration, the *UpdateAuthBlocks* function

updates ASBBs and AFBBs by checking whether each basic block's control dependencies form a subset of $\mathcal{C}_{\text{auth}}$. For instance, basic block $B7$ has a control dependency $(C4, \text{False})$. Since this branch corresponds to the authentication-success outcome recorded in $\mathcal{C}_{\text{auth}}$, $B7$ is added to $\mathcal{B}_{\text{succ}}$. Similarly, $B8$ is added to $\mathcal{B}_{\text{fail}}$. The analysis continues to iterate until no new authentication-related code elements are identified, at which point it converges and produces the final ASBBs.

D. Authentication Bypass Path Mining

Authentication bypass vulnerability detection involves identifying execution traces that reach $\mathcal{B}_{\text{succ}}$ without proper credential verification. Upon reaching any block in $\mathcal{B}_{\text{succ}}$, *AuthSpark* employs a validation module to confirm the presence of a vulnerability. Directed fuzzing efficiently explores paths toward specific targets, with $\mathcal{B}_{\text{succ}}$ serving as natural fuzzing targets. Multi-target directed fuzzing is employed with three complementary guidance strategies to systematically explore potential bypass paths.

Following existing directed-fuzzing techniques, all $\mathcal{B}_{\text{succ}}$ blocks are treated as targets, and coverage and distance metrics are employed for test prioritization. Inspired by Atropos [18], *AuthSpark* instrument string comparison functions (e.g., `strcmp` , `strncmp`, `strstr`) to capture their runtime values. The captured dynamic strings are utilized in two ways: (1) they are incorporated into a dictionary for string-based mutations, and (2) they are used as constraints for mutation generation. Together, these strategies efficiently guide fuzzing toward $\mathcal{B}_{\text{succ}}$:

1) Coverage Score. Priority scores are computed as the ratio of newly covered edges to the cumulative total number of covered edges.

2) Distance Score. For each test case, *AuthSpark* compute the control-flow distance to each $\mathcal{B}_{\text{succ}}$ block and calculate a weighted average distance, adjusting weights based on historical changes to reward progress or penalize stagnation.

3) String-guided Mutation Generation. We define *critical branches* as conditionals that prevent test cases from reaching target blocks and that, if taken differently, would enable progress toward $\mathcal{B}_{\text{succ}}$. For instance, in Figure 2, `if (v10)` at line 9 in `sub_4040` is a direct critical branch whose false branch leads to authentication success. To generate mutations that steer execution toward target basic blocks, *AuthSpark* identify critical branches by analyzing the last divergence points between the test case execution trace and the target reachable paths. To determine whether a critical branch is user-controlled, it analyze its control dependencies to identify whether it is driven directly by string-comparison operations or indirectly through function return values. For branches controlled indirectly through return values, *AuthSpark* further trace these dependencies inter-procedurally to locate the underlying string-comparison operations. Once such string comparison functions are identified, *AuthSpark* employ SMT solvers to compute string values satisfying the target branch direction, using these as mutation targets for specific input fields. In Figure 2, `strncmp` compares user input against `cfg.ua_str`, which is captured dynamically (e.g., loaded from a config file) as "WebDrive" at runtime. This string comparison determines the outcome of the condition `conn.ua_flag == 3` in `sub_4040`, thereby controlling the corresponding branch direction. By mutating the input field to satisfy this comparison, *AuthSpark* increases the probability of taking the intended branch and reaching the target basic block.

By integrating these three guidance strategies, *AuthSpark* combines coverage and distance metrics to calculate test case priorities for scheduling. Considering computational overhead, *AuthSpark* performs string-guided mutation generation only when new edge coverage is discovered.

Validation Module. During fuzzing, dynamic string mutations may cause *AuthSpark* to inadvertently generate requests that accidentally reach public endpoints not present in the original seeds. In unified authentication architectures, all requests, including both public and protected ones, are routed through the same authentication function. As a result, non-login public endpoints also trigger $\mathcal{B}_{\text{succ}}$. However, per our threat model (Section III), authentication bypass affects only protected endpoints. To eliminate false positives, *AuthSpark* employ differential testing, triggered when test cases reach any $\mathcal{B}_{\text{succ}}$ block. Specifically, the module first normalizes the test case by: (1) performing URL normalization and decoding, and (2) replacing headers with initial values since routing resides in URLs or bodies [30]. It then constructs two variants of the normalized request: one with valid authentication credentials and another with invalid credentials, and compares their responses. If both yield identical responses, the endpoint is public; otherwise, the test case represents an authentication bypass on a protected endpoint. Notably, we exclude login endpoints from this validation to ensure that test cases bypassing user credential verification (attack vector ❶ in Figure 1(b)) are not incorrectly filtered out.

VI. Implementation

We implement *AuthSpark* as a comprehensive dynamic analysis framework consisting of approximately 25,000 lines of Python code and 3,000 lines of C code. The system contains three major components:

1) Rehosting and Dynamic Instrumentation. AuthSpark operates on top of a system-level firmware rehosting environment. For each firmware sample, either FirmAE's rehosting framework [31] or the vendor's native NFV-based execution mechanism is used to boot the firmware image, which is then run on a customized QEMU-based system emulator. The emulator is built upon QEMU [32] 9.0.0 [33] running in TCG mode, where lightweight instrumentation is injected during dynamic binary translation to extract coverage feedback and memory values. The current implementation supports both x86 and ARM architectures. To collect traces on other architectures or hardware-in-the-loop devices, we implement a lightweight GDB-based tool [11] that works through the device's shell debugging interface. This tool is used solely for trace collection, as feedback-driven fuzzing relies on the instrumentation available only in our QEMU-based emulator.

2) Static Analysis Framework. The static analysis component is architecture-agnostic and is built on IDA Pro [21] and the IDALib plugin [22], [23]. *AuthSpark* constructs control-flow graphs from IDA microcode and further transform them into statement-level control- and data-dependence graphs. This framework enables execution-trace parsing, CVS detection, and ASBBs identification across multiple architectures.

3) Fuzzing Framework. The fuzzing subsystem extends ND-Fuzz [50] with HTTP protocol support, customized scheduling, and specialized mutation modules. It uses $\mathcal{B}_{\text{succ}}$ identified by *AuthSpark* as directed fuzzing targets and incorporates dynamically extracted dynamic strings as mutation dictionaries to improve mutation relevance and effectiveness.

VII. Evaluation

Research Questions. We conducted a comprehensive evaluation of *AuthSpark* on real-world embedded systems to address the following research questions:

- **RQ1:** How effective is *AuthSpark* in identifying credential verification statements and authentication-success basic blocks for detecting known vulnerabilities?
- **RQ2:** How does *AuthSpark* perform in detecting authentication bypass vulnerabilities compared to baseline fuzzing tools?
- **RQ3:** How effective is *AuthSpark* in discovering unknown vulnerabilities in real-world embedded systems?

A. Evaluation Setup

Dataset. To evaluate *AuthSpark*, we constructed a firmware dataset following two criteria: ❶ Nine firmware samples were selected because they had known vulnerabilities. These vulnerabilities should be present in the binary and accompanied by detailed information, either publicly available or manually confirmed by us, such as triggering paths and exploitation details, enabling us to validate their root causes. ❷ To further validate the effectiveness of *AuthSpark*, 23 additional firmware samples were selected from the FirmAE [31] testing set or prior research [5], [30], [51], [52] by applying three filtering steps to remove redundancy: (1) keeping only one version per device series per vendor, (2) retaining images with distinct authentication frameworks, and (3) ensuring successful full-system emulation.

In total, 32 firmware samples from 12 vendors were included in the dataset, covering 14 known vulnerabilities [1]. As shown in the 2nd and 3rd columns of Table I (details in Table VII), our dataset covers eight device types and four architectures. It consists of 32 devices, including 23 routers, two NAS devices, two access points, and five devices of other types. All of the firmware samples are rehosted using either FirmAE or manual fixes to enable system-level emulation, and configured to ensure proper access to the target web services. Due to limitations in instrumentation or emulation (e.g., incomplete boot or unsupported ISA features), two complementary methods were developed to collect execution traces: (1) QEMU-based instrumentation, which covers 22 devices with nine known vulnerabilities, referred to as dataset-1, and (2) a lightweight GDB-based tracing tool, which covers ten additional devices with five known vulnerabilities, referred to as dataset-2.

TABLE I: Basic information of the dataset. The devices using QEMU-based instrumentation and their related known vulnerabilities are depicted in the 6th and the 7th columns. The devices using GDB-based tracing and their related known vulnerabilities are depicted in the 8th and the 9th columns.

Vendor	Type	Architecture	#Device	#1-day	QEMU		GDB	
					#Device	#1-day	#Device	#1-day
DLink	Router/ IPCamera	ARMLE/ MIPS(BE/LE)	5	3	1	0	4	3
TPLINK	Router	ARMLE	2	0	2	0	0	0
Fortigate	Firewall	X86	1	1	1	1	0	0
F5	ADC	X86	1	3	1	3	0	0
Ivanti	Gateway	X86	1	1	1	1	0	0
QNAP	NAS	X86	1	1	0	0	1	1
Trendnet	Bridge/ Router	ARMLE	2	0	2	0	0	0
NETGEAR	Router/AP	ARMLE/ MIPSLE	11	2	10	1	1	1
Zyxel	NAS	ARMLE	1	1	1	1	0	0
belkin	Router	MIPSLE	1	0	0	0	1	0
ASUS	Router	ARMLE/ MIPSLE	4	2	3	2	1	0
Linksys	Router	MIPSLE	2	0	0	0	2	0
Total	**8**	**4**	**32**	**14**	**22**	**9**	**10**	**5**

Input of *AuthSpark*. After rehosting the firmware, *AuthSpark* requires pairs of authentication-success and authentication-failure requests as input. In our experiments, *AuthSpark* is evaluated on each device's default web service, yet it can also be applied to other services that implement authentication mechanisms. We manually captured requests that could successfully pass authentication and those that failed authentication as authentication request pairs. During the login authentication stage, valid or invalid admin/password pairs were used to capture authentication request pairs containing user credentials. When accessing protected resources after successful login authentication, authentication requests containing token credentials that were correct and those that were manually modified to contain incorrect token values were captured. As a result, two credential types, user credentials and token credentials are both considered.

All evaluation were conducted on an Ubuntu 20.04 with Intel Xeon Platinum 8358 (64 cores) and 512GB RAM.

B. RQ1: Identification of Authentication Related Code

To evaluate the effectiveness of *AuthSpark* in identifying CVSs and ASBBs, the dataset consists of both dataset-1 and dataset-2. Among the 32 firmware samples, 12 samples employ two credential types, whereas 18 samples use only one type due to design choices or because authentication is handled by non-binary programs (details in the 4th column of Table VI in the Appendix). Based on the definition of CVS and the credential types employed in the dataset, there are in total 44 CVSs whose locations were manually confirmed. In these 44 CVSs, 11 were confirmed to be bypassed by the 14 known vulnerabilities[2]. It should be noted that identifying

[1]The *AuthSpark* DATA-site [1] provides detailed information on the dataset along with the complete input and output data used in our evaluation.

[2]There is one CVS that is bypassed by two vulnerabilities, and another CVS bypassed by another three vulnerabilities.

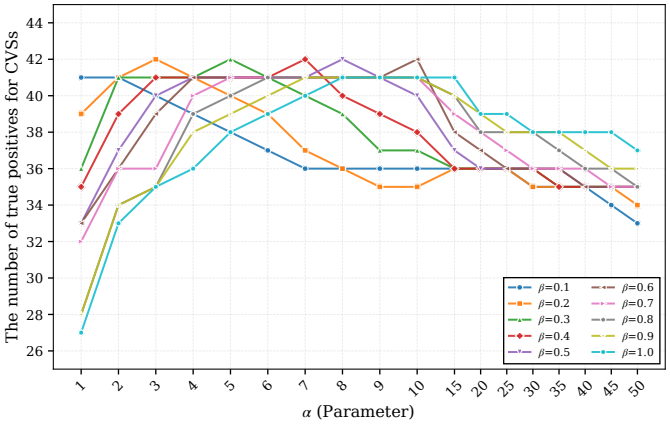

Fig. 5: CVS identification results under different parameter combinations (α, β). Each line represents a different β value ranging from 0.1 to 1.0.

TABLE II: CVS identification results of *AuthSpark* and Weasel. The "#V-CVS" column reports the number of CVSs bypassed by known vulnerabilities, with the number of vulnerabilities listed in parentheses. The "TP" column reports the number of correctly identified CVSs for each tool.

Vendor	#CVS	#V-CVS	*AuthSpark*		Weasel	
			TP	#V-CVS	TP	#V-CVS
DLink	8	2 (3)	8	2 (3)	2	0 (0)
TPLINK	3	0 (0)	2	0 (0)	1	0 (0)
Fortigate	2	1 (1)	1	1 (1)	0	0 (0)
F5	1	1 (3)	1	1 (3)	0	0 (0)
Ivanti	2	1 (1)	2	1 (1)	0	0 (0)
QNAP	2	1 (1)	2	1 (1)	1	1 (1)
Trendnet	2	0 (0)	2	0 (0)	0	0 (0)
NETGEAR	12	2 (2)	12	2 (2)	10	2 (2)
Zyxel	1	1 (1)	1	1 (1)	0	0 (0)
belkin	1	0 (0)	1	0 (0)	0	0 (0)
ASUS	7	2 (2)	7	2 (2)	2	0 (0)
Linksys	3	0 (0)	3	0 (0)	0	0 (0)
Total	**44**	**11 (14)**	**42**	**11 (14)**	**16**	**3 (3)**

a bypassed CVS is necessary to discover an authentication bypass vulnerability. The 2nd and 3rd columns of Table II illustrate these CVSs.

Credential Verification Statement Identification.

Hyperparameter. To determine appropriate scoring parameters for CVS identification, a semi-dense sweep is first conducted over the parameter space in Equation 1, evaluating all combinations of $\alpha \in \{1, \ldots, 50\}$ and $\beta \in \{0.1, \ldots, 1.0\}$. In this experiment, only the top-ranked CVS reported by *AuthSpark* in each configuration is selected to compare with the corresponding ground truth, and identified as a true positive if they are the same. The best-performing parameter combination from this sweep is then used for the rest of the paper.

Figure 5 demonstrates the necessity of tuning the reward coefficient α and penalty weight β. In summary, *AuthSpark* achieves peak performance at five parameter combinations including $(3, 0.2)$, $(5, 0.3)$, $(7, 0.4)$, $(8, 0.5)$, and $(10, 0.6)$, each achieving the same 42 correct CVSs. The worst case is $(1, 1)$, at which only 27 out of 44 CVSs (61.4%) are identified. Such results indicate that both parameters must remain within a moderate range to yield effective similarity scoring. In the rest of the paper, *AuthSpark* adopts $(3, 0.2)$ as the default configuration. In addition, all 44 CVSs are ranked at top-4 candidate CVSs identified by *AuthSpark*, further indicating the effectiveness of *AuthSpark*.

TABLE III: Ablation configurations for identifying ASBBs.

Experiment Mode	Authentication variables			Authentication conditional statements	
	Control dependency	Copy propagation	Function return variable	Variable-based	Dynamic trace-based
AuthSpark	✓	✓	✓	✓	✓
AuthSpark-NoCopy	✓	×	✓	✓	✓
AuthSpark-NoFunc	✓	✓	×	✓	✓
AuthSpark-NoVarCond	✓	✓	✓	×	✓
AuthSpark-NoDynamicCond	✓	✓	✓	✓	×

Comparison. Both *AuthSpark* and Weasel [13] aim to identify credential verification code, so the comparison is conducted between them. For a fair comparison, we reimplemented Weasel's core algorithm to support firmware analysis, otherwise it could only deal with user-mode applications. Similarly to *AuthSpark*, only the top-ranked output in Weasel's candidate list is considered the result. In addition, Weasel identifies authentication validation functions rather than individual CVSs. To address this discrepancy, we labeled a Weasel report as a true positive if the function contains at least one correct CVS.

The comparison results are presented in the 4th to 6th columns of Table II. With the configuration $(\alpha, \beta) = (3, 0.2)$, *AuthSpark* correctly identified 42 out of the 44 CVSs (95.5%), whereas Weasel identified only 16 out of 44 (36.4%). Moreover, *AuthSpark* successfully identified all 11 CVSs associated with the 14 known vulnerabilities, while Weasel detected only three of them. Detailed CVS identification results for all devices, including per-device breakdowns, are provided in Appendix Table VI.

Two CVSs are missed by *AuthSpark* because they do not rank in top-1 positions. Both of them have extremely long execution traces, which challenges the design assumption of *AuthSpark*. The long execution trace makes the code differences induced by the actual authentication logic occupy only a small fraction, making *AuthSpark* more likely to misrank them. The deep analysis is described in Appendix Section B.

In contrast, Weasel's decision-tree–based identification and the large number of divergence points make it ineffective. For the failure cases, Weasel tends to incorrectly report either the callers or the callees of the true authentication validation functions, or even completely unrelated functions, exhibiting three types of false positives. The detailed analysis is described in Appendix Section C.

Authentication-success Basic Block Identification.

To validate the effectiveness of our authentication-success basic block identification, we evaluated the ability of *AuthSpark* and its variants to analyze firmware samples. The variants are constructed to selectively disable the rules involved in identifying $\mathcal{V}_{\text{auth}}$ and $\mathcal{C}_{\text{auth}}$. Table III summarizes these configurations. The control-dependency rule is retained in all variants because it forms the foundation of *AuthSpark* for two reasons: (1) Authentication variables serve as the

TABLE IV: For each vendor, the table reports the number of authentication-related code elements identified under different ablation configurations. “#1-day” indicates the number of known vulnerabilities whose triggering paths cover at least one authentication-success basic block in $\mathcal{B}_{\text{succ}}$. Detailed per-device results are provided in Table VII.

Vendor	*AuthSpark*				*AuthSpark*-NoCopy				*AuthSpark*-NoFunc				*AuthSpark*-NoVarCond				*AuthSpark*-NoDynamicCond			
	$\#\mathcal{V}_{\text{auth}}$	$\#\mathcal{C}_{\text{auth}}$	$\#\mathcal{B}_{\text{succ}}$	#1-day	$\#\mathcal{V}_{\text{auth}}$	$\#\mathcal{C}_{\text{auth}}$	$\#\mathcal{B}_{\text{succ}}$	#1-day	$\#\mathcal{V}_{\text{auth}}$	$\#\mathcal{C}_{\text{auth}}$	$\#\mathcal{B}_{\text{succ}}$	#1-day	$\#\mathcal{V}_{\text{auth}}$	$\#\mathcal{C}_{\text{auth}}$	$\#\mathcal{B}_{\text{succ}}$	#1-day	$\#\mathcal{V}_{\text{auth}}$	$\#\mathcal{C}_{\text{auth}}$	$\#\mathcal{B}_{\text{succ}}$	#1-day
DLink	39	25	28	3	20	14	15	1	15	13	15	1	21	8	12	1	36	22	25	3
TPLINK	9	6	6	0	5	2	2	0	7	5	5	0	9	3	3	0	9	6	6	0
Fortigate	41	59	69	1	9	17	24	0	2	3	3	0	12	3	3	0	41	59	69	1
F5	5	3	2	3	2	1	1	0	2	1	1	0	2	1	1	0	5	3	2	3
Ivanti	9	5	12	1	6	4	6	0	3	2	4	0	8	5	6	0	8	5	10	1
QNAP	13	77	100	1	7	45	67	1	1	3	3	0	10	3	4	0	12	76	96	1
Trendnet	5	5	9	0	4	5	8	0	1	4	5	0	3	2	3	0	5	5	9	0
NETGEAR	49	67	89	2	30	51	71	0	34	56	76	0	16	12	22	0	49	65	87	2
Zyxel	4	3	5	1	2	1	2	1	1	1	2	0	3	1	2	1	4	3	5	1
belkin	6	3	3	0	5	2	2	0	3	2	2	0	6	1	1	0	6	3	3	0
ASUS	38	27	51	2	16	16	28	1	9	12	13	0	21	12	16	0	37	21	43	2
Linksys	14	8	9	0	5	4	4	0	2	3	3	0	11	4	4	0	13	7	8	0
Total	**232**	**288**	**383**	**14**	**111**	**162**	**230**	**4**	**80**	**105**	**132**	**1**	**122**	**55**	**77**	**2**	**225**	**275**	**363**	**14**

initial carriers through which authentication state propagates during analysis, and (2) All authentication variables inferences, including function-return variables, require considering their control-dependency relationship with $\mathcal{C}_{\text{auth}}$. Moreover, the known vulnerabilities associated with the identified ASBBs are also evaluated. For each known vulnerability, its payload is executed to determine whether the triggering path covers any basic block in the identified $\mathcal{B}_{\text{succ}}$ set. If so, the vulnerability is considered to be potentially exposed.

The results are summarized in Table IV. The full version of *AuthSpark*, which enables all identification rules, discovers most authentication-related elements, identifying 383 ASBBs and exposing all 14 known vulnerabilities. Furthermore, the ablation variants reveal the contribution of individual rules: ❶ *AuthSpark-NoCopy*: disabling the copy-propagation rule yields only 230 identified ASBBs (60.1% of the full version) and exposes only four vulnerabilities. ❷ *AuthSpark-NoFunc*: disabling the function return variable rule results in only 132 identified ASBBs (34.5% of the full version), exposing only one vulnerability. ❸ *AuthSpark-NoVarCond*: removing the variable-based ACS rule produces the fewest ASBBs (77 blocks) and exposes only two vulnerabilities. ❹ *AuthSpark-NoDynamicCond*: disabling the dynamic trace-based ACS rule reduces the number of identified ACSs and ASBBs (13 and 20 fewer than full *AuthSpark*, respectively). Notably, disabling such a rule would result in the discovery of one fewer 0-day vulnerability (referring to Report Issue 1 in Table V), even though it did not impact the number of known vulnerabilities in the experiment.

Disabling any identification rule not only reduces the number of elements in the corresponding category but also decreases related elements. For the two AV-related variants, the number of AVs decreases by 121 and 152, respectively, which also leads to a reduction of 126 and 183 ACSs. This demonstrates that function-return variables constitute a substantial portion of AVs, and disabling this rule greatly diminishes the propagation of authentication state to ACSs and ASBBs. For the two ACS-related variants, the number of ACSs decreases by 233 and 13, which in turn reduces the number of AVs by 110 and 7. This shows that the variable-based ACSs identification rule plays a central role in propagating authentication state, and removing it significantly reduces both AVs and ASBBs.

Overall, the ablation results clearly demonstrate that removing any major identification rule substantially weakens *AuthSpark*'s identification capability. Compared with the full *AuthSpark* configuration, the variants *AuthSpark*-NoCopy, *AuthSpark*-NoFunc, and *AuthSpark*-NoVarCond exhibit a reduction by a factor of 1.67–4.97 in the number of ASBBs (383 vs. 230, 132, and 77), which directly leads to a significant decline in their ability to expose the known vulnerabilities. Detailed ASBBs identification results for all variants, including per-device breakdowns, are provided in Appendix Table VII.

C. RQ2: Fuzzing for Authentication Bypass Vulnerability Detection

To evaluate the effectiveness of *AuthSpark* in detecting authentication bypass vulnerabilities, its performance is compared against two ablation variants and three existing web service fuzzing tools:

- *AuthSpark-NoStr*, which disables the provision of dynamic strings for fuzzing.
- *AuthSpark-NoStr-NoMutate*, which further removed string-guided mutation generation, retaining only basic fuzzing capabilities.
- *SRFuzzer* [49], which specializes in fuzzing web services on embedded systems.
- *Snipuzz* [14], which performs response-guided fuzzing for embedded systems.
- *BooFuzz* [25], which is a general-purpose protocol fuzzer supporting various network protocols.

Since existing fuzzers do not originally support detecting authentication bypass vulnerabilities, they are equipped with the ability to monitor the execution of basic blocks in $\mathcal{B}_{\text{succ}}$, which in turn enables the fuzzers to detect such vulnerabilities. As fuzzing requires QEMU-based instrumentation support, dataset-1 that contains 22 firmware samples was used, which covers nine known vulnerabilities. Each firmware sample was fuzzed for 24 hours and the process was repeated five times.

Detailed detection results of authentication bypass vulnerabilities are presented in Table V, where the rows labeled with CVE identifiers present the detection results for all known vulnerabilities. *AuthSpark* detected all nine known vulnerabilities. Among the two ablated variants, *AuthSpark-NoStr* de-

TABLE V: Authentication bypass vulnerability detection of *AuthSpark*. Column abbreviations: AS = *AuthSpark*; AS-NS = *AuthSpark*-NoStr; AS-NSM = *AuthSpark*-NoStr-NoMutate; SR = SRFuzzer; SN = Snipuzz; BO = BooFuzz.

Vendor	#Fuzz Devices	Bug ID	AS	AS-NS	AS-NSM	SR	SN	BO
DLink	1	Report Issue 1	✓	✓	✓	✓	✓	✓
TPLink	2	-	×	×	×	×	×	×
Fortigaete	1	CVE-2022-40684	✓	×	×	×	×	×
F5	1	CVE-2020-5902	✓	×	×	×	×	×
		CVE-2021-22986	✓	✓	✓	×	×	×
		CVE-2022-1388	✓	✓	✓	×	×	×
Ivanti	1	CVE-2023-46805	✓	✓	×	×	×	×
Trendnet	2	-	×	×	×	×	×	×
NETGEAR	10	Report Issue 2 (PSV-2025-0044)	✓	×	×	×	×	×
		Report issue 3 (PSV-2025-0044)	✓	×	×	×	×	×
		CVE-2021-34977	✓	✓	×	×	×	×
Zyxel	1	CVE-2023-4473	✓	×	×	×	×	×
		Report Issue 4 (CVE-2024-6342)	✓	×	×	×	×	×
ASUS	3	CVE-2021-32030	✓	✓	✓	×	×	×
		CVE-2021-20090	✓	✓	×	×	×	×
		Report Issue 5 (CVE-2025-2492)	✓	✓	✓	✓	✓	✓
		Report Issue 6 (CVE-2025-59366)	✓	✓	×	×	×	×
Total	**22**	**#1-day**	**9**	**6**	**3**	**0**	**0**	**0**
		#0-day	**6**	**3**	**2**	**2**	**2**	**2**

```
1  int sub_1A5C(int a1, int a2){
2      ...
3      st_samba_mode = nvram_get_st_samba_mode(v5);
4      if ( st_samba_mode == 1 && !strstr(url, "aware")){
5          strcpy(a2->username, "guest");
6          strcpy(a2->passwd, &unk_5608);
7      }
8      else{
9          if ( do_account_authentication(username, passwd) != 1 )
10             return 2;  // auth failure ✗
11     }
12     if(!j)
13         j = malloc(50);
14     ...
15     return 1;          // auth success ✓
16 }
```

Fig. 6: Code snippet for case study.

tected only six known vulnerabilities, and *AuthSpark*-*NoStr*-*NoMutate* detected three known vulnerabilities, demonstrating the critical importance of both dynamic string extraction and specialized mutation strategies. None of the remaining three fuzzers, SRFuzzer, Snipuzz, and BooFuzz, detected any of the known vulnerabilities.

Case Study. It is worth noting that although CVE-2021-34977 is a known 1-day authentication bypass vulnerability, no public technical details are available. During fuzzing of the NETGEAR XR300, *AuthSpark* independently rediscovered this vulnerability. Our analysis shows that the bypass is triggered inside the authentication routine: any request whose URL contains the substring "rpc" bypasses the CVS in function `sub_39BC4` and directly triggers the execution of an ASBB.

D. RQ3: Unknown Vulnerability Discovery

All fuzzers were applied to detect 0-day authentication bypass vulnerabilities in the latest firmware versions of the devices in dataset-1. As shown in Table V, the rows marked "Report Issue" indicate the 0-day authentication bypass vulnerabilities detected during fuzzing. Using guided fuzzing, *AuthSpark* successfully discovered six 0-day vulnerabilities. Among the two ablated variants, *AuthSpark*-*NoStr* detected only three 0-day vulnerabilities, and *AuthSpark*-*NoStr*-*NoMutate* detected two. The remaining three fuzzers detected two 0-day vulnerabilities. All six 0-day vulnerabilities detected by *AuthSpark* have been verified on the latest firmware versions and confirmed by the respective vendors, with four identifiers assigned, including three CVE IDs and one PSV ID.

Case Study. All fuzzers detected Report Issue 1 and Report Issue 5 by monitoring the ASBBs identified by *AuthSpark*. ❶ *Report Issue 1.* In the DIR665 firmware, a weak authentication check is performed for a subset of post-authentication requests, accepting them as long as the request appears to originate from the MAC address of a previously authenticated device. ❷ *Report Issue 5.* As illustrated in Figure 6, when the Samba service is enabled by administrators, specific URL requests can bypass authentication entirely. The vulnerability arises because enabling Samba mode causes the code execution trace to have the same effect as successful authentication (i.e., `do_account_authentication` returns 1). *AuthSpark* successfully identified the ASBBs within this function, enabling the detection of the vulnerability.

VIII. Discussion

Extensibility of *AuthSpark*. The core design scope of *AuthSpark* also imposes several inherent limitations. Certain authentication implementation patterns fall outside the behaviors *AuthSpark* is designed to model, which may hinder its analysis and prevent full reconstruction of the authentication logic. ❶ Authentication state may be stored in unconventional carriers, such as in Juniper SRX [26] where the authentication result is also written to a session file and later verified only through file contents. *AuthSpark* does not yet support tracking such diverse forms of authentication-state storage, which we leave for future work. ❷ Static-analysis limitations may hinder *AuthSpark*'s handling of complex data structures. In cases where the authentication result resides in a structure accessed through multi-level pointers, static analysis may fail to propagate the state beyond the current basic block, preventing complete tracking. ❸ Authentication logic may span multiple modules, as in SonicWall GMS [39] whose processing involves both Apache and Tomcat. *AuthSpark* can currently analyze the authentication logic within each module individually, but stitching these partial analyses into a complete end-to-end authentication flow remains future work.

Although *AuthSpark* focuses on detecting authentication bypass vulnerabilities in back-end binary programs written in C/C++ for embedded web services, the proposed method can be extended to analyze authentication models in other types of services and implementations written in different programming languages. This paper provides researchers with a clearer understanding of the implementation of authentication modules,

facilitating more detailed analysis of potential vulnerabilities therein.

Threat to Validation. First, *AuthSpark* only supports backend programs developed with C and C++. Second, the precision of the disassembly engine may impact the outcomes of control flow and data flow analysis, thus affecting trace-to-pseudocode alignment accuracy. We implemented best-effort alignment strategies and observed no impact on $\mathcal{B}_{\text{succ}}$ identification across all evaluated cases. Third, there may be errors in the ground truth manually confirmed.

Responsible Disclosure. We have responsibly disclosed all vulnerabilities we found. We provided detailed information and PoC to the vendors for each vulnerability, facilitating their confirmation and reproduction of the vulnerabilities.

IX. Conclusion

In this paper, we presented *AuthSpark*, a novel dynamic analysis technique for detecting authentication bypass vulnerabilities. *AuthSpark* identifies authentication-success basic blocks by analyzing execution traces and enhances effectiveness through static analysis. These blocks serve as targets for directed fuzzing, enabling automated discovery of bypass paths. *AuthSpark* correctly locates 42 out of 44 credential verification statements, detects all 14 known vulnerabilities, and uncovers six zero-day flaws, with three CVEs and one PSV assigned.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is partly supported by Chinese National Natural Science Foundation (Grants #62032010, #62202462, #62302500).

Ethics Considerations

The primary ethical consideration in our work is ensuring that the vulnerabilities identified by *AuthSpark* are responsibly disclosed. We follow the vulnerability disclosure process outlined in §VIII to notify vendors and relevant organizations [9], aiming to reduce the risk of exploitation by attackers. All vulnerabilities are confirmed in a controlled local environment, ensuring the legitimacy of the investigation. Additionally, our work does not involve human subjects, personal data, or other activities with significant ethical concerns.

References

[1] Anonymous. Authspark data. https://anonymous.4open.science/r/AuthSpark_DATA-5F17, 2025. Accessed: 2025.

[2] David Basin, Patrick Schaller, and Jorge Toro-Pozo. Inducing authentication failures to bypass credit card {PINs}. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3065–3079, 2023.

[3] Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48. IEEE, 2000.

[4] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS*, pages 1–15, 2018.

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

[6] Cisco. Active exploitation of cisco ios xe software web management user interface vulnerabilities. https://blog.talosintelligence.com/active-exploitation-of-cisco-ios-xe-software/, 2023. Accessed: 2023.

[7] Cisco. Cisco auth bypass vulnerability. https://sec.cloudapps.cisco.com/security/center/content/CiscoSecurityAdvisory/cisco-sa-iosxe-webui-privesc-j22SaA4z, 2023. Accessed: 2023.

[8] Cisco. Cisco authentication bypass vulnerability with active in-the-wild exploitation. https://github.com/Shadow0ps/CVE-2023-20198-Scanner/blob/main/Exploitation_Explainer.md, 2023. Accessed: 2023.

[9] CVE. Cve database. https://www.cve.org/, 2025. Accessed: 2025.

[10] Eric Glass Davenport. The ntlm authentication protocol and security support provider. https://curl.se/rfc/ntlm.html. Technical Specification.

[11] GDB developers. Gdb source. https://sourceware.org/gdb/, 2025. Accessed: 2025.

[12] Nguyen Minh Duc and Bui Quang Minh. Your face is not your password face authentication bypassing lenovo–asus–toshiba. *Black Hat Briefings*, 4:158, 2009.

[13] explosion. weasel github. https://github.com/explosion/weasel, 2025. Accessed: 2025.

[14] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC conference on computer and communications security*, pages 337–350, 2021.

[15] Google Cloud. Authentication overview. https://cloud.google.com/docs/authentication, 2025. Accessed: 2025.

[16] Ryan Grandgenett, William Mahoney, and Robin Gandhi. Authentication bypass and remote escalated i/o command attacks. In *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, pages 1–7, 2015.

[17] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. Symbion: Interleaving symbolic with concrete execution. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–10. IEEE, 2020.

[18] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. Atropos: Effective fuzzing of web applications for {Server-Side} vulnerabilities. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4765–4782, 2024.

[19] Songming Han, Jieke Lu, and Shaofeng Ming. Hardcoded vulnerability mining method in a simulated environment based on router backdoor detection technology. In *Proceedings of the 2024 International Conference on Generative Artificial Intelligence and Information Security*, pages 280–284, 2024.

[20] D. Hardt. The oauth 2.0 authorization framework. RFC 6749, October 2012. https://www.rfc-editor.org/rfc/rfc6749.

[21] Hex-Rays. The interactive disassembler pro is a computer software disassembler which generates assembly language code from machine-executable code. https://hex-rays.com/ida-home/, 2005.

[22] Hex-Rays. An ida plugin which makes it possible to write scripts for ida. https://www.hex-rays.com/products/ida/support/idapython_docs/, 2019.

[23] Hex-Rays. Idalib allows you to use the c++ and ida python apis outside ida as standalone applications. https://docs.hex-rays.com/user-guide/idalib, 2024.

[24] Yikun Jiang, Wei Xie, and Yong Tang. Detecting authentication-bypass flaws in a large scale of iot embedded web servers. In *Proceedings of the 8th International Conference on Communication and Network Security*, pages 56–63, 2018.

[25] jtpereyda. boofuzz github. https://github.com/jtpereyda/boofuzz, 2025. Accessed: 2025.

[26] Juniper. Juniper srx. https://www.juniper.net/gb/en/products/security/srx-series.html, 2025. Accessed: 2025.

[27] Ejin Kim and Hyoung-Kee Choi. Security analysis and bypass user authentication bound to device of windows hello in the wild. *Security and Communication Networks*, 2021(1):6245306, 2021.

[28] Lighttpd. lighttpd. https://www.lighttpd.net/, 2025. Accessed: 2025.

[29] Hangtian Liu, Shuitao Gan, Chao Zhang, Zicong Gao, Hongqi Zhang, Xiangzhi Wang, and Guangming Gao. Labrador: Response guided

directed fuzzing for black-box iot devices. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1920–1938. IEEE, 2024.

[30] Hangtian Liu, Lei Zheng, Shuitao Gan, Chao Zhang, Zicong Gao, Hongqi Zhang, Yishun Zeng, Zhiyuan Jiang, and Jiahai Yang. Eagleye: Exposing hidden web interfaces in iot devices via routing analysis. In *NDSS*, 2025.

[31] pr0v3rbs. Firmae. https://github.com/pr0v3rbs/FirmAE, 2025. Accessed: 2025.

[32] QEMU. Qemu. https://www.qemu.org/, 2025. Accessed: 2025.

[33] QEMU. Qemu 9.0.0. https://www.qemu.org/2024/04/23/qemu-9-0-0/, 2025. Accessed: 2025.

[34] Rapid7. Cve-2023-20198: Active exploitation of cisco ios xe zero-day vulnerability. https://www.rapid7.com/blog/post/2023/10/17/etr-cve-2023-20198-active-exploitation-of-cisco-ios-xe-zero-day-vulnerability/, 2023. Accessed: 2025.

[35] J. Reschke. The 'basic' http authentication scheme. Technical report, RFC 7617, 2015.

[36] Nadav Rotenberg, Haya Shulman, Michael Waidner, and Benjamin Zeltser. Authentication-bypass vulnerabilities in soho routers. In *Proceedings of the SIGCOMM Posters and Demos*, pages 68–70. 2017.

[37] Felix Schuster and Thorsten Holz. Towards reducing the attack surface of software backdoors. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 851–862, 2013.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1, 2015.

[39] Sonicwall. Sonicwall gms. https://www.sonicwall.com/products/management-and-reporting/global-management-system, 2025. Accessed: 2025.

[40] Prashast Srivastava, Hui Peng, Jiahao Li, Hamed Okhravi, Howard Shrobe, and Mathias Payer. Firmfuzz: Automated iot firmware introspection and analysis. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*, pages 15–21, 2019.

[41] Hui Jun Tay, Kyle Zeng, Jayakrishna Menon Vadayath, Arvind S Raj, Audrey Dutcher, Tejesh Reddy, Wil Gibbs, Zion Leonahenahe Basque, Fangzhou Dong, Zack Smith, et al. Greenhouse:{Single-Service} rehosting of {Linux-Based} firmware binaries in {User-Space} emulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5791–5808, 2023.

[42] Christian Tiefenau, Maximilian Häring, Mohamed Khamis, and Emanuel von Zezschwitz. " please enter your pin"–on the risk of bypass attacks on biometric authentication on mobile devices. *arXiv preprint arXiv:1911.07692*, 2019.

[43] Wikipedia contributors. Embedded system. https://en.wikipedia.org/wiki/Embedded_system, 2025. Accessed: 2025.

[44] Wikipedia contributors. Gestalt pattern matching. https://en.wikipedia.org/wiki/Gestalt_pattern_matching, 2025. Accessed: 2025.

[45] Wikipedia contributors. Lcs algorithm. https://en.wikipedia.org/wiki/Longest_common_subsequence, 2025. Accessed: 2025.

[46] Wikipedia contributors. Smt. https://en.wikipedia.org/wiki/Satisfiability_modulo_theories, 2025. Accessed: 2025.

[47] Haoyu Xiao, Ziqi Wei, Jiarun Dai, Bowen Li, Yuan Zhang, and Min Yang. Housefuzz: Service-aware grey-box fuzzing for vulnerability detection in linux-based firmware. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 3801–3819. IEEE, 2025.

[48] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, volume 14, pages 1–16, 2014.

[49] Yu Zhang, Wei Huo, Kunpeng Jian, Ji Shi, Haoliang Lu, Longquan Liu, Chen Wang, Dandan Sun, Chao Zhang, and Baoxu Liu. Srfuzzer: An automatic fuzzing framework for physical soho router devices to discover multi-type vulnerabilities. In *Proceedings of the 35th annual computer security applications conference*, pages 544–556, 2019.

[50] Yu Zhang, Nanyu Zhong, Wei You, Yanyan Zou, Kunpeng Jian, Jiahuan Xu, Jian Sun, Baoxu Liu, and Wei Huo. Ndfuzz: a non-intrusive coverage-guided fuzzing framework for virtualized network devices. *Cybersecurity*, 5(1):21, 2022.

[51] Jiaxu Zhao, Yuekang Li, Yanyan Zou, Zhaohui Liang, Yang Xiao, Yeting Li, Bingwei Peng, Nanyu Zhong, Xinyi Wang, Wei Wang, and Wei Huo. Leveraging semantic relations in code and data to enhance taint analysis of embedded systems. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 7067–7084, Philadelphia, PA, August 2024. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/zhao.

[52] Jiaxu Zhao, Yuekang Li, Yanyan Zou, Yang Xiao, Naijia Jiang, Yeting Li, Nanyu Zhong, Bingwei Peng, Kunpeng Jian, and Wei Huo. From constraints to cracks: Constraint semantic inconsistencies as vulnerability beacons for embedded systems. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, August 2025. https://www.usenix.org/conference/usenixsecurity25/presentation/zhao.

[53] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

APPENDIX

A. LCS Optimization for Trace Analysis

To mitigate the computational complexity of applying LCS to lengthy execution traces, our approach employs call-stack-based segmentation to divide traces into manageable units.

Call Stack Information. Through trace analysis, we recover the call stack information at each function call sites, obtaining tuples of (call stack, trace index). For example, as shown in Figure 2, the call to do_account_authentication yields the tuple ([sub_4040, basic_authentication, 0x4218], 59123), where "0x4218" is the call site address within basic_authentication and "59123" is the execution index in the address-level trace.

Alignment Point Identification. From each trace, we extract the sequence of call stacks at all function call sites, each with its trace index. Consider two traces with call stack sequences (call site addresses omitted for clarity), where A and B–E denote functions and $\rightarrow$ represents the call relationship:

- Trace 1: $[(A, 100), (A \rightarrow B, 250), (A \rightarrow C, 550), (A \rightarrow D, 850), ...]$
- Trace 2: $[(A, 120), (A \rightarrow B, 270), (A \rightarrow C, 380), (A \rightarrow E, 620), ...]$

where each tuple contains (call stack, trace index). Applying LCS to the call stack portions identifies the common subsequence $[A, A \rightarrow B, A \rightarrow C]$ with length 3. Assuming a context window threshold of 3 (this value can be empirically adjusted based on the required matching context between execution sequences), these matching entries become alignment points: $(100, 120)$, $(250, 270)$, and $(550, 380)$. These alignment point pairs indicate which trace positions should be aligned, effectively segmenting the traces into comparable units for efficient divergence detection.

Trace Segmentation. Using these alignment points, we partition the lengthy execution traces into smaller segments. Within each segment pair, we further divide the traces into multiple units and efficiently apply LCS to identify divergence points without computational overhead.

The necessity of this optimization is demonstrated in our experiments: when disabled, the analysis crashed during differential point extraction on ASUS RT-AC68U, which contains over 70,000 basic blocks in its execution trace. This segmentation technique enables our system to handle such large-scale traces that would otherwise be computationally infeasible.

B. False-Positive Analysis of CVS Identification in AuthSpark

As shown in Table VI, the correct CVS for both FortiOS and TP-Link Archer C8 does not appear in the top-1 of *AuthSpark* 's ranked results. The underlying causes for these two cases are similar, and we take FortiOS as a case study to illustrate this issue. In FortiOS, the full execution traces of the authentication request pairs span more than 50,000 basic blocks, while the true authentication logic affects only a very small portion of these traces. Because the correct CVS introduces only a short-lived divergence in the successful and failed traces, the two traces quickly converge again and continue to share a long common prefix. As a result, when a later conditional statement introduces another divergence, the long common prefix shared by the two traces keeps its prefix similarity extremely high (e.g., 0.99), while the suffix similarity becomes much lower (0.12 vs. 0.66 at the correct CVS). Under Equation 1, this combination yields a higher overall score for the unrelated statement, causing *AuthSpark* to incorrectly rank it above the correct CVS.

This false positive illustrates a general challenge: in extremely long traces, when the code differences induced by the actual authentication logic occupy only a small fraction of the total execution, the differential signal becomes overshadowed by other incidental divergence points, making misranking more likely.

C. False-Positive Analysis of CVS Identification in Weasel

In our comparison, Weasel correctly identifies 16 authentication-verification functions using its top-ranked candidate for each device. Weasel identifies credential-verification code by constructing decision trees from multiple request traces and selecting the dominator nodes as candidate validation functions. For the failure cases, Weasel exhibits three types of false positives:

- **Type I (15 cases):** No call-relationship to the true validation function. These occur because when the number of divergence points is large, different divergence points may share similar dominator-like structural patterns, causing unrelated nodes to be mistakenly ranked as candidates.
- **Type II (13 cases):** Functions that are callers of the correct validation function. These arise because Weasel tends to select functions that are invoked more frequently across requests, making common callers appear as high-ranked candidates.
- **Type III (2 cases):** Functions that are callees of the correct validation function.

Overall, these results show that relying solely on decision-tree–derived dominator patterns is fundamentally insufficient for accurately identifying credential-verification logic.

D. Detailed Results of CVSs Identification

Table VI summarizes the detailed results of CVS identification. The table reports not only the top-1 accuracy but also the accuracy behavior under different (α, β) settings, illustrating how the similarity-scoring parameters influence identification quality. In particular, it includes the hyperparameter study where $\alpha = 3$ is fixed and β is varied from 1 to 0.1 (parameter settings that yield identical results are not shown), showing how different β values affect the ranking performance. All intermediate and final outputs produced by *AuthSpark* and Weasel during their detection procedures, as well as the exact firmware versions of each evaluated device, are available on the *AuthSpark*_DATA site [1].

E. Detailed Results of ASBBs Identification

Table VII presents the detailed results of ASBBs identification, along with comprehensive information about the firmware

TABLE VI: The detailed top-1 results of credential verification code identification. The notation (α, β) denotes a set of hyperparameter pairs with $\alpha = 3$, arranged to show an overall increasing trend in CVS identification. The table reports the CVS identification results for *AuthSpark* and Weasel. The "#Diff" column reports the number of divergence points. The last column lists the function in which each credential verification point resides, i.e., the authentication function. The "Weasel Ranking" column shows the ranking produced by Weasel and the position of the correct authentication function in that ranking.

Vendor	Model	WEB Type	Credential Type	#Diff	(3,1)	(3,0.7)	(3,0.6)	(3,0.5)	(3,0.4)	(3,0.2)	(3,0.1)	Weasel	Weasel Ranking	Weasel's Auth Function	Auth Function
DLink	DCS-930L	alphapd	User	12	×	×	×	✓	✓	✓	✓	×	6/14	websUrlProcessRequest	websCheckRealm
	DIR-300	cgibin	User	3	✓	✓	✓	✓	✓	✓	✓	×	-/1	sessioncgi_main	authentication
			Token	4	✓	✓	✓	✓	✓	✓	✓	×	-/5	servicecgi_main	sub_40819C
	DIR-505L	User	lighttpd	1	✓	✓	✓	✓	✓	✓	✓	✓	1/6	do_login	do_login
	DIR-665	httpd	User	3	✓	✓	✓	✓	✓	✓	✓	✓	1/2	sub_EE6C	sub_EE6C
			Token	10	✓	✓	✓	✓	✓	✓	✓	×	5/5	sub_15244	sub_B818
	DIR-882	goahead	User	2	✓	✓	✓	✓	✓	✓	✓	×	3/3	sub_424090	sub_42141C
			Token	5	✓	✓	✓	✓	✓	✓	✓	×	11/23	sub_423DF4	sub_422FDC
TPLINK	Archer_C3150	httpd	User	4	✓	✓	✓	✓	✓	✓	✓	✓	1/6	sub_13524	sub_13524
			Token	10	×	×	✓	✓	✓	✓	✓	×	4/10	sub_D22C	sub_D898
	Archer_C8	httpd	User	20	×	×	×	×	×	×	×	×	-/31	sub_1EA54	sub_C393C
Fortigate	FortiOS	httpd	User	23	×	×	×	×	×	×	×	×	-/44	sub_AA2850	sub_230A950
			Token	13	×	×	✓	✓	✓	✓	✓	×	-/61	sub_AD5750	sub_216A2B0
F5	BIGIP	apache	Token	42	✓	✓	✓	✓	✓	✓	✓	×	3/29	log_error_core	sub_59B0
Ivanti	Ivanti Connect Secure	httpd	VPN-Token	26	✓	✓	✓	✓	✓	✓	×	×	4/121	sub_97200	sub_9B660
			RestAPI-Token	27	✓	✓	✓	✓	✓	✓	✓	×	17/33	sub_F48B0	sub_F3E20
QNAP	TS-231P	thttpd	User	7	×	×	×	×	✓	✓	✓	×	-/13	main	Check_Local_User_Password
			Token	3	✓	✓	✓	✓	✓	✓	✓	✓	1/3	auth_get_session	auth_get_session
Trendnet	TEW-828DRU	httpd	User	6	✓	✓	✓	✓	✓	✓	✓	×	3/4	sub_1479C	sub_13DD8
	TEW800	httpd	User	2	×	×	✓	✓	✓	✓	✓	×	2/14	sub_AAFC	sub_B888
NETGEAR	WNR3500	httpd	User	4	×	×	×	×	×	✓	✓	✓	1/7	sub_AF90	sub_AF90
	XR300	httpd	User	24	✓	✓	✓	✓	✓	✓	✓	✓	1/7	sub_39BC4	sub_39BC4
	EX6200	httpd	User	3	✓	✓	✓	✓	✓	✓	✓	✓	1/7	sub_1C67C	sub_1C67C
			Token	33	✓	✓	✓	✓	✓	✓	✓	×	-/11	sub_DD1C	sub_1C0C4
	R6200V2	httpd	User	11	✓	✓	✓	✓	✓	✓	✓	✓	1/14	sub_ED50	sub_ED50
	R6300V2	httpd	User	17	✓	✓	✓	✓	✓	✓	✓	✓	1/9	sub_EE74	sub_EE74
	R6400v2	httpd	User	18	✓	✓	✓	✓	✓	✓	✓	✓	1/10	sub_10DC4	sub_10DC4
	R6700V3	mini-httpd	User	21	✓	✓	✓	✓	✓	✓	✓	✓	1/56	sub_103DC	sub_103DC
	R7000	mini-httpd	User	18	×	✓	✓	✓	✓	✓	✓	✓	1/15	sub_1123C	sub_1123C
	R7000P	mini-httpd	User	23	✓	✓	✓	✓	✓	✓	✓	×	2/32	sub_19F00	sub_10F3C
	R8000	httpd	User	16	✓	✓	✓	✓	✓	✓	✓	✓	1/8	sub_10694	sub_10694
	WAC104	mini-httpd	User	4	✓	✓	✓	✓	✓	✓	✓	✓	1/13	sub_406ADC	sub_406ADC
Zyxel	NAS326	apache	Token	3	✓	✓	✓	✓	✓	✓	✓	×	-/42	ap_send_error_response	sub_17A4
belkin	F7D4301	httpd	User	1	✓	✓	✓	✓	✓	✓	✓	×	2/3	ht_SetLoginIP	sub_42D4F8
ASUS	RT-AC68U	lighttpd	User	39	✓	✓	✓	✓	✓	✓	✓	×	-/31	stat_cache_get_entry	sub_2F0C
			Token	37	✓	✓	✓	✓	✓	✓	✓	×	46/145	sub_4040	do_account_authentication
	RT-AX56U	mini-httpd	User	6	✓	✓	✓	✓	✓	✓	✓	✓	1/10	sub_30518	sub_30518
			Token	17	✓	✓	✓	✓	✓	✓	✓	×	10/58	sub_5FFDC	sub_5B478
	DSL-AC88U	mini-httpd	User	5	✓	✓	✓	✓	✓	✓	✓	✓	1/7	sub_222CC	sub_222CC
			Token	38	✓	✓	✓	✓	✓	✓	×	×	-/67	sub_199C0	sub_1D9F8
	RT_N10	mini-httpd	User	2	✓	✓	✓	✓	✓	✓	✓	×	2/3	sub_40507C	sub_40425C
Linksys	E1000	httpd	User	1	✓	✓	✓	✓	✓	✓	✓	×	-/4	set_login_info	login_check
			Token	5	✓	✓	✓	✓	✓	✓	✓	×	-/2	set_accept_language	sub_41CFE4
	WRT320N	httpd	User	2	✓	✓	✓	✓	✓	✓	✓	×	5/6	send_headers	sub_41BC44
					35	36	39	40	41	42	40	16			

samples in the *AuthSpark* evaluation dataset, including device model, device type, web server type, and architecture. The "Trace" column indicates how execution traces were collected, either via GDB or QEMU. For architectures such as MIPS or for hardware-in-the-loop devices where QEMU instrumentation is unavailable, GDB was used as an alternative to obtain the required traces.

Notably, as observed from the detailed per-device results in Table VII, FortiOS and QNAP TS-231P exhibit exceptionally large numbers of ASBBs. This significant difference stems from their request-processing architecture. These devices first parse the URL to determine which handler function should process the request, and authentication-protected URLs have their authentication checks embedded within their respective handler functions. This results in multiple authentication function calls distributed across different URL handlers. In contrast, other devices call the authentication function once at the beginning, then dispatch to different handlers based on the URL. Static variable propagation is particularly effective for the former architecture as it can trace authentication outcomes across all distributed authentication function calls.

F. Vulnerability Detection Examples During Fuzzing

During fuzzing, whenever any basic block in the ASBBs set is executed and subsequently validated as a true positive by

TABLE VII: Basic information of the tested devices and the detailed results of recognizing authentication-related code elements across *AuthSpark* and its ablation variants. A vulnerability is considered detected if its triggering path reaches any block in the authentication-success basic-block set ($\mathcal{B}_{\text{succ}}$). The last two columns report the number of identified 1-day and 0-day authentication-bypass vulnerabilities, respectively. Column abbreviations: AS = *AuthSpark* ; A1 = *AuthSpark*-NoCopy; A2 = *AuthSpark*-NoFunc; A3 = *AuthSpark*-NoVarCond; A4 = *AuthSpark*-NoDynamicCond.

Model	**Arch**	**Trace**	**DevType**	$\mathcal{V}_{\text{auth}}$					$\mathcal{C}_{\text{auth}}$					$\mathcal{B}_{\text{succ}}$					$\mathcal{B}_{\text{fail}}$					**1-day #vuln**					**0-day #vuln**				
				AS	A1	A2	A3	A4	AS	A1	A2	A3	A4	AS	A1	A2	A3	A4	AS	A1	A2	A3	A4	AS	A1	A2	A3	A4	AS	A1	A2	A3	A4
DCS-930L	MIPSLE	GDB	IPCamera	5	2	3	2	5	6	3	3	1	5	6	3	4	3	5	5	3	3	2	5	0	0	0	0	0	0	0	0	0	0
DIR-300	MIPSLE	GDB	Router	22	6	3	8	19	10	4	4	2	8	10	4	4	2	8	10	3	3	2	8	0	0	0	0	0	0	0	0	0	0
DIR-505L	MIPSBE	GDB	Router	3	3	3	3	3	2	2	2	1	2	2	2	2	2	2	2	2	2	2	2	0	0	0	0	0	0	0	0	0	0
DIR-665	ARMLE	QEMU	Router	4	4	4	4	4	2	2	2	2	2	3	3	3	3	3	2	2	2	2	2	0	0	0	0	0	1	1	1	1	0
DIR-882	MIPSLE	GDB	Router	5	5	2	4	5	5	3	2	2	5	7	3	2	2	7	5	3	2	2	5	3	1	1	1	3	0	0	0	0	0
Archer_C3150	ARMLE	QEMU	Router	6	3	4	6	6	5	1	4	2	5	5	1	4	2	5	4	1	3	2	4	0	0	0	0	0	0	0	0	0	0
Archer_C8	ARMLE	QEMU	Router	3	2	3	3	3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
FortiOS	X86	QEMU	Firewall	41	9	2	12	41	59	17	3	3	59	69	24	3	3	69	55	15	2	2	55	1	0	0	0	1	0	0	0	0	0
BIGIP	X86	QEMU	ADC	5	2	2	2	5	3	1	1	1	3	2	1	1	1	2	2	1	1	1	2	3	0	0	0	3	0	0	0	0	0
Ivanti Connect Secure	X86	QEMU	Gateway	9	6	3	8	8	5	4	2	5	5	12	6	4	6	10	10	5	2	4	10	1	0	0	0	1	0	0	0	0	0
TS-231P	X86	GDB	NAS	13	7	1	10	12	77	45	3	3	76	100	67	3	4	96	74	44	2	3	74	1	1	0	0	1	0	0	0	0	0
TEW-828DRU	ARMLE	QEMU	Router	4	3	0	2	4	2	2	1	1	2	5	4	1	2	5	2	2	1	1	2	0	0	0	0	0	0	0	0	0	0
TEW800	ARMLE	QEMU	MediaBridge	1	1	1	1	1	3	3	3	1	3	4	4	4	1	4	3	3	3	1	3	0	0	0	0	0	0	0	0	0	0
WNR3500	ARMLE	QEMU	Router	2	1	2	2	2	2	1	2	1	2	2	1	2	1	2	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0
XR300	ARMLE	QEMU	Router	13	7	5	1	13	14	5	6	1	12	19	9	9	2	17	14	4	6	1	14	1	0	0	0	1	0	0	0	0	0
EX6200	ARMLE	QEMU	AP	8	2	1	3	8	5	3	2	2	5	5	3	2	2	5	5	3	2	2	5	0	0	0	0	0	0	0	0	0	0
R6200V2	ARMLE	QEMU	Router	2	2	2	1	2	4	4	4	1	4	5	5	5	1	5	4	4	4	1	4	0	0	0	0	0	0	0	0	0	0
R6300V2	ARMLE	QEMU	Router	5	4	5	1	5	6	5	6	1	6	6	5	6	1	6	4	4	4	1	4	0	0	0	0	0	0	0	0	0	0
R6400v2	ARMLE	QEMU	Router	4	3	4	1	4	9	8	9	1	9	12	11	12	2	12	9	9	9	1	9	0	0	0	0	0	0	0	0	0	0
R6700V3	ARMLE	QEMU	Router	6	4	6	1	6	7	6	7	1	7	10	8	10	3	10	3	3	3	1	3	0	0	0	0	0	1	0	1	0	1
R7000	ARMLE	QEMU	Router	1	1	1	1	1	2	2	2	1	2	4	4	4	2	4	1	1	1	1	1	0	0	0	0	0	1	1	1	1	1
R7000P	ARMLE	QEMU	Router	1	1	1	1	1	6	6	6	1	6	8	8	8	1	8	4	4	4	1	4	0	0	0	0	0	0	0	0	0	0
R8000	ARMLE	QEMU	Router	4	3	4	1	4	11	10	11	1	11	12	11	12	1	12	5	5	5	1	5	0	0	0	0	0	0	0	0	0	0
WAC104	MIPSLE	GDB	AP	3	2	3	3	3	1	1	1	1	1	6	6	6	6	6	1	1	1	1	1	1	0	0	0	1	0	0	0	0	0
NAS326	ARMLE	QEMU	NAS	4	2	1	3	4	3	1	1	1	3	5	2	2	2	5	1	1	1	1	1	1	1	0	1	1	1	0	0	0	1
F7D4301	MIPSLE	GDB	Router	6	5	3	6	6	3	2	2	1	3	3	2	2	1	3	3	2	2	1	3	0	0	0	0	0	0	0	0	0	0
RT-AC68U	ARMLE	QEMU	Router	17	5	1	8	17	13	8	4	5	11	26	15	4	6	22	11	7	2	4	11	0	0	0	0	0	2	1	0	0	2
RT-AX56U	ARMLE	QEMU	Router	9	6	2	5	8	7	5	4	3	5	15	7	4	4	13	10	5	3	2	9	1	1	0	0	1	0	0	0	0	0
DSL-AC88U	ARMLE	QEMU	Router	8	4	4	4	8	4	2	2	2	3	6	4	3	4	5	3	2	1	2	3	1	0	0	0	1	0	0	0	0	0
RT_N10	MIPSLE	GDB	Router	4	1	2	4	4	3	1	2	2	2	4	2	2	2	3	3	2	1	1	2	0	0	0	0	0	0	0	0	0	0
E1000	MIPSLE	GDB	Router	8	3	1	6	8	4	2	2	2	4	5	2	2	2	5	4	2	2	2	4	0	0	0	0	0	0	0	0	0	0
WRT320N	MIPSLE	GDB	Router	6	2	1	5	5	4	2	1	2	3	4	2	1	2	3	2	1	1	2	3	0	0	0	0	0	0	0	0	0	0
Total				232	111	80	122	225	288	162	105	55	275	383	230	132	77	363	263	146	80	52	260	14	4	1	2	14	6	3	3	2	5

the verification module, *AuthSpark* records the corresponding test case. Each example includes the complete fuzzing request as well as the ASBBs blocks triggered by the authentication-bypass payload. A subset of fuzzing-detection examples for known vulnerabilities is provided on the *AuthSpark*_DATA site [1].

G. Effectiveness of Validation Module.

We also analyzed the effectiveness of the validation module in eliminating false alarms caused by public endpoints triggering $\mathcal{B}_{\text{succ}}$, with the most significant example being in the ASUS RT-AX56U. In this case, the module reduced alerts from 2,689 to 158 within 24 hours, with all filtered requests confirmed to be accessing non-login public endpoints. This targeted filtering approach maintains high precision while preserving the detection capability for critical authentication vulnerabilities.