Your Firmware Has Arrived: A Study of Firmware Update Vulnerabilities

Yuhao Wu[†], Jinwen Wang[†], Yujie Wang[†], Shixuan Zhai[†],
Zihan Li[†], Yi He[§], Kun Sun[‡], Qi Li[§], Ning Zhang[†]

[†] *Washington University in St. Louis,*
[§] *Tsinghua University,* [‡] *George Mason University*

Abstract

Embedded devices are increasingly ubiquitous in our society. Firmware updates are one of the primary mechanisms to mitigate vulnerabilities in embedded systems. However, the firmware update procedure also introduces new attack surfaces, particularly through vulnerable firmware verification procedures. Unlike memory corruption bugs, numerous vulnerabilities in firmware updates stem from incomplete or incorrect verification steps, to which existing firmware analysis methods are not applicable. To bridge this gap, we propose ChkUp, an approach to Check for firmware Update vulnerabilities. ChkUp can resolve the program execution paths during firmware updates using cross-language inter-process control flow analysis and program slicing. With these paths, ChkUp locates firmware verification procedures, examining and validating their vulnerabilities. We implemented ChkUp and conducted a comprehensive analysis on 12,000 firmware images. Then, we validated the alerts in 150 firmware images from 33 device families, leading to the discovery of both zero-day and n-day vulnerabilities. Our findings were disclosed responsibly, resulting in the assignment of 25 CVE IDs and one PSV ID at the time of writing.

1 Introduction

The rapid increase of embedded devices, ranging from portable devices, such as smartwatches, to large machines like transport vehicles, brings more connectivity and convenience to our lives. The market size of embedded devices is expected to reach 116.2 billion dollars in 2025 [10]. Firmware update plays an important role in fixing vulnerabilities and improving functionalities. However, a poorly implemented firmware update mechanism can diminish the benefit or even introduce new attack surfaces. In fact, vulnerabilities related to software update has been recognized as one of the top five security risks for embedded devices [12].

Firmware Update Security. Software or firmware updates are currently one of the most effective techniques against cyber attacks. However, with the increasing connectivity and complexity in modern embedded systems, there is an increasing number of cyber attacks targeting specifically the firmware update procedures, allowing the adversary to execute arbitrary code or roll back the firmware version to expose prior vulnerabilities [21, 70]. Recent vulnerabilities in update mechanisms of Jeep Cherokee [52], Samsung SmartThings Hub [8], and Asus Router [9] have raised significant concerns, highlighting the need for automatic identification of firmware update vulnerabilities. Existing firmware vulnerability detection methods often focus on identifying invocations to unsafe sinks from user-controlled inputs [19, 25, 27, 59, 65] or finding bugs using common vulnerable patterns or deviations from known specifications [24, 33, 49, 50, 64]. However, firmware update vulnerabilities pose a unique challenge as they often arise from a combination of issues across multi-stage update procedures, exacerbated by the absence of comprehensive specifications or systematic categorizations of vulnerable patterns. To gain a better understanding of the landscape of firmware update vulnerabilities, we categorize and systematize firmware update-related CVEs in the past decade. Each type of vulnerability is placed in different abstract phases of a general firmware update procedure, which is detailed in Section 2.

Our Solution - ChkUp. In this paper, we propose ChkUp, a novel approach to Check for firmware Update vulnerabilities, including missing verification (e.g. a lack of version check) and improper verification (e.g., use of MD5 for integrity check). Intuitively, ChkUp extracts the program execution paths for a firmware update procedure and then identifies the chain of verification steps in the update procedure. We then summarize the vulnerable patterns across multiple firmware update phases for vulnerability detection. While working on this, we found that existing implementations of firmware update mechanisms present unique challenges that require new techniques. Specifically, there are three main technical challenges:

C1. Diverse System Components Supporting Firmware Update: In many Linux-based firmware images, various types of

programs such as front-end programs, scripts, and binaries are invoked in the software update execution path that spans both the front end and back end of the web server. Additionally, the diversity of components involved in firmware updates leads to the heterogeneity of inter-process communication (IPC) mechanisms. To tackle these challenges, we develop techniques to generate the inter-process update flow graph (UFG). The entry program is first identified using common code patterns and semantic information that interconnects the front end and the back end in a firmware update procedure. Then, cross-language control flows are extracted by connecting the control flows of individual programs (i.e., front-end programs, scripts, and binaries) and resolving the corresponding IPCs. With such a cross-language cross-program control flow graph, firmware update execution paths are resolved using backward program slicing with firmware update-specific semantics.

C2. Verification Procedure Recognition: A firmware update is a complex procedure that includes a variety of checks and verifications, from the verification of cryptographic signatures to the comparison of versions and device IDs. An update procedure is considered secure only when every verification step (e.g., signature verification or compliance with version update policies) and their composition are properly implemented. However, firmware updates do not have standardized specifications and are often implemented in multiple programming languages. This leads to diverse verification implementations, making it challenging to identify them from numerous functions in execution paths. To tackle this challenge, ChkUp recognizes firmware verification procedures using data flow graph (DFG) isomorphism-based semantic similarity matching. To reduce the overhead of this process, functions in execution paths are first ranked by a similarity score, which is calculated using both syntactic and structural features. Then, functions with higher similarity scores are prioritized for DFG isomorphism-based analysis.

C3. Vulnerability Validation: Static analysis can produce a number of false positives (FP), which require further validation. However, a firmware update procedure often involves a chain of verification steps, each with its unique functions and invocation parameters. To test a step later in the chain, it is necessary to create an input and an environment that are capable of passing the first several steps. To simplify validation, we present a semi-automated dynamic method to validate alerts for emulatable firmware images. Specifically, we employ firmware patching to ensure the execution of potentially vulnerable procedures. The validity of the corresponding alerts is then checked based on the update behaviors after inputting malicious firmware images.

Evaluation and Findings. To gain a deeper understanding of vulnerabilities in the wild, we ran ChkUp[1] on 12,000 firmware images. We found that weak verification algorithms, such as the use of MD5 for integrity verification, were prevalent.

Then, we performed vulnerability validation on 150 randomly selected firmware images: For those emulatable firmware images, we performed dynamic validation by creating PoCs; for the remaining firmware images, we conducted manual analysis to validate their alerts. Our results showed a true positive rate (TPR) of 86.7% and a false positive rate (FPR) of 5.3%, leading to the discovery of both zero-day and n-day vulnerabilities in firmware images from 33 device families. These findings were responsibly disclosed, and 25 CVE IDs and one PSV ID have been assigned. Finally, to demonstrate the exploitability of the vulnerabilities, we showcased firmware downgrade and firmware modification attacks.

Contributions. Our contributions are outlined as follows:

- *A systematization of firmware update security:* We frame general firmware update procedures into four phases and examine security issues in each phase by analyzing and categorizing 381 firmware update-related CVE reports.
- *A new approach for update vulnerability detection:* We propose a new firmware update vulnerability identification approach, ChkUp, which addresses three technical challenges: diverse components in update paths, verification procedure recognition, and vulnerability validation.
- *Vulnerabilities on real-world firmware:* We ran ChkUp on 12,000 firmware images and validated alerts for 150 of them using a combination of proof-of-concept (PoC) generation and manual analysis. The results demonstrate ChkUp's capability to identify zero-day and n-day vulnerabilities. Following responsible disclosure, 25 CVE IDs and one PSV ID have been assigned.

2 Firmware Update Security Systematization

2.1 Challenges of Firmware Updates

To gain a better understanding of the threat landscape, 381 firmware update-related CVEs from the past decade are analyzed and then systemized. While most CVEs stem from vulnerable firmware update mechanisms, others simply have an impact on the security of the update. Figure 1 shows the annual distribution of Common Vulnerability Scoring System (CVSS) v3 metrics since 2015. There is a steady trend of increasing, with the number of CVEs doubling from 2020 to 2021. High and critical vulnerabilities are nearly
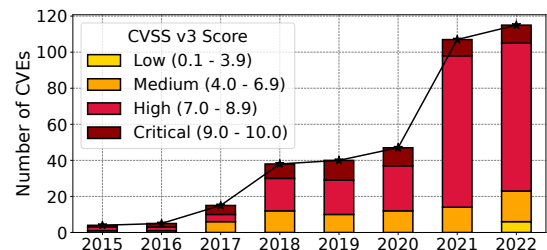

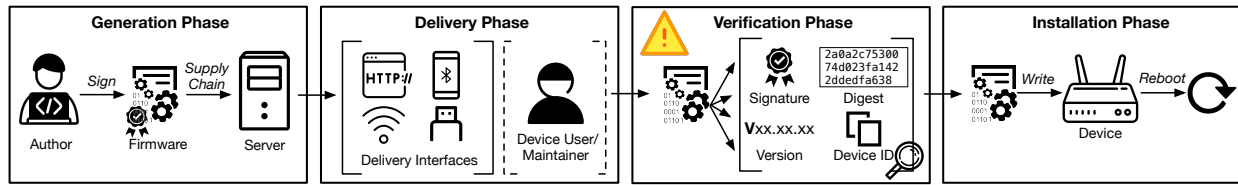Figure 1: CVSSv3 scores of firmware update-related CVEs.

Figure 2: A common firmware update workflow.

four times of low and medium ones. This increase can be attributed to not only the general trend of increasing CVEs but also the newly arising challenges specific to embedded systems [15, 30, 62, 72–74, 84, 86]. These challenges include 1) an expanding attack surface due to increased connectivity, 2) increased complexity of embedded systems, 3) long product life cycle, and 4) limited resources on embedded devices. These factors contribute to diverse firmware update mechanisms and increased vulnerabilities.

2.2 Update Workflow and Vulnerabilities

A typical firmware update workflow has four phases as shown in Figure 2: generation, delivery, verification, and installation.

Generation Phase. The goal of the generation phase is to create firmware images and make them available. Specifically, an author first develops a new firmware image and a manifest. The manifest contains firmware metadata, including the firmware digest, version, and device ID. subsequently, the firmware image and manifest are signed with a digital signature and then transferred to a firmware server through the software supply chain. Usually, the author uploads firmware to the server through trusted parties in the supply chain.

During this phase, external attackers can conduct supply chain attacks to steal certificates and compromise software development tools or infrastructure. The root cause of such attacks is inappropriate access control for critical assets and infrastructures. There has been an increasing number of supply chain-related vulnerabilities reported in recent years, demonstrating their security impact in the real world. Reports indicate that the U.S. Government and more than 30,000 public and private organizations such as Microsoft, Intel, and Fire-Eye suffered from a large-scale software supply chain attack known as the SolarWinds hack in 2020 [55]. Specifically, cybercriminals compromised Orion IT management software and then distributed malicious software updates containing backdoors to users through the supply chain.

Takeaway 1: Supply chain vulnerabilities pose a significant risk, often from inadequate access controls. Without proper on-device verification, compromised firmware can be installed in devices, leading to a loss of control over them.

Delivery Phase. The delivery phase involves transmitting the new firmware image from the server to the target device. A

firmware component, known as the *update agent* [46], handles downloading, verifying, and storing the new image in persistent memory. Typically, new firmware can be delivered in three ways: 1) The firmware server directly pushes the firmware and manifest to the device's *update agent*; 2) The *update agent* polls for updates and downloads them when they become available; 3) The firmware server notifies the device user/maintainer, who manually downloads and uploads updates to the *update agent*. In terms of communication channels, common methods include application-layer protocols (e.g., HTTP, FTP), wireless media (e.g., Wi-Fi, Bluetooth Low Energy), and physical interfaces (e.g., USB, removable memory cards). For some low-end, bare-metal devices, companion apps on smartphones can assist with firmware updates. The new firmware can either be bundled within or fetched by the app from the firmware server through application-layer protocols. Then, these apps generally communicate with the device via wireless media for notification, polling, and downloading. It is worth noting that while the apps act as intermediaries for transferring firmware, they may also pre-verify updates. Such early verification can filter out invalid updates, to avoid unnecessary, subsequent on-device processing.

A secure communication channel is important for the confidentiality and integrity of the delivered firmware images. Insecure delivery mainly arises from a lack of cryptographic protocols or the use of hard-coded keys, exposing systems to machine-in-the-middle (MITM) attacks. For example, CVE-2020-9544 involves plain HTTP without authentication, while CVE-2020-25233 derives from the use of a hard-coded RSA key for communication. Mobile apps can also have insecure communication with either firmware servers or devices, as shown by CVE-2018-3928, where insufficient communication security checks can lead to code execution vulnerabilities. Importantly, the primary concern is not just the communication channel but also the lack of proper security verification. For example, even with a leaked communication key, if a robust firmware verification mechanism is in place, malicious firmware replacements during updates can be prevented.

Takeaway 2: Firmware delivery security mainly relies on the communication channel and device user/maintainer. If either is insecure, the device may receive compromised firmware unless proper on-device verification is in place.

Verification Phase. The verification phase ensures the authenticity, integrity, freshness, and compatibility of the received

Table 1: Top ten firmware update-related CWE vulnerability types.

CWE ID	Name	Example	Proportion
CWE-345	Insufficient Verification of Data Authenticity	CVE-2020-10831, CVE-2020-24395, CVE-2022-36360	5.47%
↪CWE-347	Improper Verification of Cryptographic Signature	CVE-2020-27540, CVE-2021-37160, CVE-2022-21134	10.16%
CWE-287	Improper Authentication	CVE-2018-6294, CVE-2020-27488, CVE-2022-2503	6.25%
↪CWE-306	Missing Authentication for Critical Function	CVE-2019-16243, CVE-2020-9544, CVE-2020-29379	3.91%
↪CWE-295	Improper Certificate Validation	CVE-2018-15476, CVE-2020-15498, CVE-2021-22909	3.13%
CWE-20	Improper Input Validation	CVE-2018-3891, CVE-2019-11103, CVE-2021-25437	10.94%
CWE-434	Unrestricted Upload of File with Dangerous Type	CVE-2019-10959, CVE-2021-37160, CVE-2022-28372	5.47%
CWE-798	Use of Hard-coded Credentials	CVE-2019-5158, CVE-2019-14926, CVE-2020-24215	2.34%
CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	CVE-2017-11082, CVE-2017-14444, CVE-2017-14445	1.56%
CWE-639	Improper Neutralization of Special Elements used in an OS Command	CVE-2018-3890, CVE-2019-5157, CVE-2019-15310	0.78%

Note: CWE-347 is the child class of CWE-345; CWE-306 and CWE-395 are the child classes of CWE-287.

firmware. Specifically, the *update agent* performs a series of verification procedures before storing the image in persistent memory: firmware authenticity is ensured by verifying the digital signature of the firmware; firmware integrity is verified by checking the digest contained in the manifest; and the freshness and compatibility are confirmed by examining the metadata, along with version and device ID in the manifest.

Table 1 lists the top ten firmware update-related vulnerabilities in the Common Weakness Enumeration (CWE) category, based on our CVE analysis. The top eight categories, accounting for 47.67%, predominantly involve either *missing verification* or *improper verification* methods for firmware updates. These issues can enable attackers to replace the benign firmware with a malicious one during updates. For instance, the issue with CVE-2018-10988 stems from a lack of digital signature verification in the shell script used for firmware updates. Missing or improper integrity verification may lead to firmware corruption. For example, using easily bypassable internal checksums for firmware integrity checks is problematic (e.g., CVE-2018-5441). Missing or improper freshness verification can lead to firmware downgrade attacks, while inadequate compatibility verification can expose the device to DoS attacks. For instance, the root cause of CVE-2018-3891 is a logic flaw in performing version verification, where integer comparison operators are incorrectly used for string comparison. Similarly, in the case of CVE-2020-10831, arbitrary firmware can be installed due to insufficient verification.

> **Takeaway 3:** Either missing or improper implementation of any steps in the verification procedure can lead to the installation of unintended firmware on the embedded device.

Installation Phase. The installation phase is a process of installing and executing the new firmware. After verification, the new firmware is stored in the persistent memory of the device and is activated upon reboot. Specifically, a bootloader first moves the new firmware image to the right offset in the device memory when the device is starting up. Then, the bootloader executes the new firmware image after conducting a firmware inspection. However, this inspection is often incomplete and insecure, commonly relying on internal checksums [46].

Most vulnerabilities in this phase are typical software bugs such as command injection and memory corruption bugs.

Specifically, firmware update-related commands executed during this phase may accept parameters from user inputs. If an attacker manipulates these parameters and they are subsequently used by vulnerable functions (e.g., *system*, *strcpy*), it can lead to command injection (e.g., CVE-2019-5155) or memory corruption (e.g., CVE-2021-22675) attacks.

> **Takeaway 4:** Incomplete firmware inspection procedures in bootloaders during firmware installation are common, thus making the security of firmware updates dependent on the verification mechanisms in the *update agent*.

Summary. Security vulnerabilities can arise during any phase of the firmware update process. Nevertheless, robust firmware verification mechanisms by the device's *update agent* can mitigate the majority of vulnerabilities originating from other phases. Hence, our research primarily focuses on identifying vulnerabilities within the verification phase.

3 Threat Model and Overview

Threat Model. ChkUp aims to uncover firmware update vulnerabilities in OS-based firmware (integrated with file systems), particularly in the dominant Linux-based firmware [69]. It can detect the most prevalent firmware update vulnerabilities including missing or improper verification of authenticity, integrity, freshness, and compatibility. Aligned with existing research [19, 34, 36, 59, 83], we assume no firmware source code access, making ChkUp a binary-based vulnerability detection approach. Potential users of ChkUp could be security researchers seeking to notify vendors, or end-users trying to obtain additional security information about their devices. Even vendors with access to the source code can benefit, especially in investigating the exploitability of vulnerabilities, since source code analysis can overlook binary and runtime-level details. It is worth noting that, similar to intrusion detection systems or malware detectors, in-depth domain expertise proves valuable in further refining alerts.

Overview of ChkUp. The high-level idea of ChkUp is to statically extract the firmware update program execution paths from the firmware codebase and to pinpoint potential vulnerabilities along these paths based on summarized vulnerability
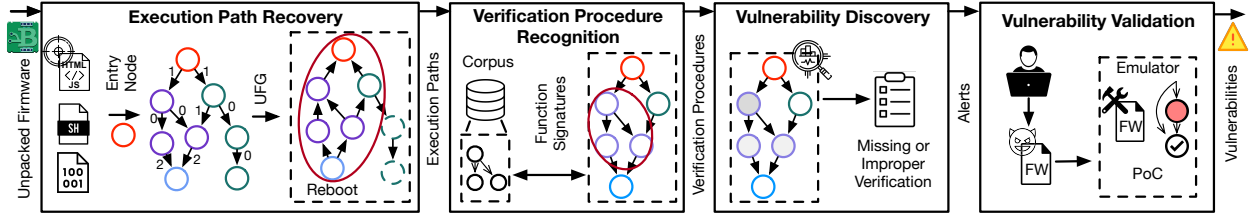
Figure 3: Overview of ChkUp.

patterns. Then, dynamic vulnerability validation is performed to reduce false alerts. However, as discussed in Section 1, three primary challenges need to be addressed to implement this idea: *C1. Diverse Programs in Update Paths* during the extraction of firmware update execution paths, *C2. Verification Procedure Recognition* for matching vulnerability patterns, and *C3. Vulnerability Validation* to reduce false alerts. To address these challenges, we propose ChkUp (illustrated in Figure 3). Specifically, to address *C1*, we first create a UFG that captures the control flow information across programs written in different programming languages. Next, we perform backward program slicing to determine the firmware update execution paths (Section 4.1). To tackle *C2*, we extract syntactic and structural features for function matching, then employ more sophisticated DFG isomorphism to recognize the verification chains in the firmware update execution paths (Section 4.2). With the execution paths and the associated verification procedures, we examine them to discover vulnerabilities based on defined criteria (Section 4.3). Finally, we address *C3* by a patching-based method where the vulnerable procedure is tested using the generated PoCs after its execution dependencies are bypassed via patching (Section 4.4).

4 Design of ChkUp

4.1 Execution Path Recovery

UFG Definition. The binary dependency graph (BDG) in Karonte [59] can model data dependencies between binaries within firmware images, which is crucial for firmware update vulnerability detection. However, accurate firmware update execution path recovery requires additional information, including control flows, IPCs, and program invocations across programs in various languages to determine the intra- and inter-process control flows of firmware update-related programs. Therefore, building upon BDG, we introduce UFG to accommodate these requirements. A UFG, represented by G, is a directed graph that captures the intra- and inter-process control flow information at the basic block (BB) level of firmware update-related programs. The UFG is defined as $G = (V, E)$, where V is a set of BBs extracted from front-end programs, scripts, and binaries involved in the firmware update procedure. E represents the directed

edges between the BBs, and each edge $e \in E$ is represented as $e = ([v_1, p_1], [v_2, p_2], c)$. This indicates that BB v_1 in program p_1 either transfers execution flow or shares data with BB v_2 in program p_2, and c is a flag indicating the type of edge: c is 0 for intra-process control flow edges, c is 1 for IPC relations, and c is 2 for program invocation relations.

Update Entry Finding. Receiving firmware is typically the first step in an update procedure on the device side. Therefore, the entry node in a UFG is the node responsible for this task, and the program that includes this entry node is referred to as the entry program. For firmware containing a web-based update interface, the entry program could be a front-end firmware upload utility. To identify the entry program, we use a static pattern-matching approach. This is non-trivial as firmware update mechanisms vary greatly among different vendors. To address this issue, we manually analyze numerous firmware images (details are provided in Appendix B.2) and identify distinct patterns that differentiate firmware update entry programs from others. Specifically, such an entry program always contains recognizable code patterns. For example, front-end programs to upload firmware images might use the *<input type="file"...>* pattern, while scripts or binaries that download firmware images might use the *wget ...* pattern. Additionally, the entry program often displays prompt messages containing common informative words, as well as function and variable names (e.g., *fw_version* and *fw_upload*) related to firmware updates. Built upon these observations, we identify the program that matches the highest number of predefined patterns as the entry program.

Cross-language Control Flow Analysis. After identifying the entry program, the next step is to locate the programs for processing the received firmware image. These programs can take different forms, such as binaries and shell scripts [48]. To gain complete insight into the control flow of the programs executed during a firmware update, cross-language control flow analysis is necessary. However, current path exploration-based vulnerability detection methods [19, 59] lack this capacity. To address this challenge, we interpret the control flow logic, IPC paradigms, and program invocation paradigms of various program types (i.e., HTML paired with JavaScript, shell script, and binary) commonly used in firmware updates to construct UFGs. Specifically, we build the call graph (CG) and inter-procedure control flow graph (CFG) of the entry

```
FUN_0001aa68 in binary_a
01.void FUN_0001aa68(undefined4 param_1,FILE *param_2,
uint param_3){...
02.   // IPC set function invocaton
03.   nvram_set_int("upgrade_fw_status",0);...
04.   // library function invocation
05.   pcStack60 = "/sbin/ejusb";...
06.   // program execution function invocation
07.   _eval(&pcStack60,0,0,0);...
nvram_get in script_a
01.// IPC get function invocation
02.var upgrade_fw_status = '<% nvram_get
("upgrade_fw_status"); %>';...
_start in binary_b
01.void _start(undefined4 param_1){...
02.   // main function invocation
03.   __uClibc_main(FUN_0000e998,in_stack_00000000,
&stack0x00000004,_init,_fini,param_1);...}
```

Listing 1: IPC and program invocation examples.

program and identify IPC (i.e., sockets, files, signals, environment variables, NVRAM, and shared memory) and program invocation paradigms in the entry program. Note that the supported program types and IPC paradigms are determined by our preliminary manual firmware analysis, and their generalizability is empirically measured using a large number of firmware images. Then, dependency edges representing control dependencies between the entry program and other related programs are created. The procedure is repeated in a recursive manner, where the CFG of each newly added program is built, and IPC and program invocation paradigms are identified. Listing 1 illustrates the connection of two programs through the discovery of IPC paradigms (*nvram*) and program invocation paradigms (*eval*) between them. The UFG construction is completed once no more update-related programs are found.

Function-level Backward Slicing. With the constructed UFG, the next step is to determine the possible program execution paths for a firmware update procedure using function-level backward program slicing. Typically, a firmware update procedure concludes with a reboot to execute the new firmware. Thus, we locate calls to the reboot function, especially those that trigger the *reboot* binary, in the UFG and set them as the target of backward slicing. The backward slicing follows the inter-procedural control flow within and across programs to determine all possible execution paths starting from the firmware receiving function and ending with the reboot function. These paths represent potential firmware update execution paths. However, as with any path-based exploration analysis, this execution path recovery method may encounter the issues of path explosion and path missing. We tackle path explosion with five strategies, including skipping standard libraries, skipping built-in utilities, using a timeout, filtering paths with prompts, and merging paths. Further details on these strategies can be found in Appendix A.1.

4.2   Verification Procedure Recognition

Two-stage Approach Overview. Key functions are those used in firmware verification, such as hash functions for in-

```
01.int divideUpLoadFile(char *file_name,UPGRADE_FILE_HEADER
*pFileHeader){...
02.iVar2 = md5_verify_digest(fileMd5Checksum,input,uVar4);
03.if (iVar2 == 0){
04.   pcVar5 = "[%s:%d]md5 error\n"; ...}...}
05.
06.int md5_verify_digest(uchar *digest,uchar *input,int len){
07.   int iVar1;
08.   uchar digst [16];
09.   md5_make_digest(digst,input,len);
10.   iVar1 = memcmp(digst,digest,0x10);
11.   return (uint)(iVar1 == 0);}
```

Listing 2: An integrity verification procedure example.

tegrity verification. Our manual analysis reveals that verification procedures often use a similar set of key functions since they all try to accomplish a similar set of functions. Moreover, values from functions, either as return variables or pass-by-reference arguments, often flow into conditional expressions. These expressions then influence which conditional branch is taken. An example can be seen when verifying firmware integrity: the return value of a hash function is used in a conditional expression to determine the verification result, as demonstrated in Listing 2. To identify verification procedures, we construct a corpus of function signatures from common key functions. Next, we recognize verification procedures from the execution paths in two stages: efficient function similarity matching, which quickly filters out irrelevant functions; and verification procedure chain identification, where we use advanced semantic analysis for precise identification.

Efficient Function Similarity Matching. The goal of the first stage is to improve the efficiency of the identification of verification procedure chains by ranking functions with syntactic and structural features. Kim et al. [39] show that the use of numeric syntactic and structural features can efficiently achieve comparable accuracy to more complex deep learning-based methods. Therefore, a combination of numeric features (see Table 4 in Appendix A.2) is selected for two reasons: 1) These features can be efficiently extracted without requiring complex semantic code analysis; 2) The combination of these features can achieve high discriminative capability while being robust across different architectures, compiler types, and code stripping. It is worth noting that the impact of compiler optimizations on these features becomes less significant in embedded systems, where stable and industry-standard compiler options are commonly used, thereby ensuring the robustness of features. Specifically, we obtain syntactic features from the intermediate representation (IR) of code, including attributes from its abstract syntax tree (AST) such as the number of function calls and key strings. Structural features come from CFGs, including the number of BBs, edges, and branches. Using these extracted features, similarity scores between functions in execution paths and those in the corpus are calculated based on the relative difference between their feature values [39]. These scores improve efficiency in the second stage by prioritizing functions with higher similarity scores. Function pairs with similarity scores below a specific threshold (0.5 in this work) are filtered due to the low like-

lihood of similarity. More details on the extracted features, mathematical construction of similarity scores, and similarity score threshold determination can be found in Appendix A.2.

Verification Procedure Chain Identification. A data-flow graph (DFG) can represent the relationship between the data and the arithmetic and logical operations, and it is used in cryptographic primitive identification techniques [47, 51]. To further identify key functions with accurate semantics analysis, we employ DFG subgraph isomorphism. In addition to function semantics, the usage pattern of the return variables and pass-by-reference arguments of key functions is also considered to reduce the FPs of verification procedure matching. The insight is that the return variables or arguments from a key function often feed into a conditional expression. This expression then determines the conditional branch to execute and indicates the pass or fail status of the verification procedure. An example can be seen in Listing 2, where the return variable of the function *md5_verify_digest* is used to determine whether the integrity verification has passed.

We search for verification procedure chains from all execution paths. For each execution path, various types of verification procedures are identified successively. To identify verification procedures, such as authenticity verification, we first select a function pair (f, f') with the highest similarity score, where f is from the execution path and f' is a key function for authenticity verification in the corpus. Then, the DFG of the function f is constructed and normalized to preserve its underlying semantics while eliminating variations introduced by developers, compiler optimizations, or machine code translation. With the DFG, we can verify whether the functions f and f' are functionally equivalent by searching for subgraphs in the DFG of f that are isomorphic to the graph signature of f', using Ullmann's algorithm [71]. If a match is found, function f is considered semantically equivalent to function f'. A reaching-definition analysis is then performed on the return variable and pass-by-reference arguments of f to obtain data-flow slices. If the return variable or arguments flow into conditional expressions, the verification procedure is considered identified. The process continues with the next function pair having the next highest similarity score if no match is found unless there are no remaining function pairs exceeding the similarity score threshold. Upon completion of the second stage, verification procedure chains in all execution paths are identified and ready to be further examined.

Corpus Creation. Key functions are either standard library functions or proprietary functions. Functions responsible for authenticity and integrity checks often encapsulate standard cryptographic routines from well-established libraries [85]. To this end, we collect open-source cryptographic functions from libraries commonly used in embedded systems. Though functions used for freshness and capability checks are usually proprietary, they share similarities within the same device family based on our analysis. These are obtained through reverse engineering of firmware images from various major vendors. We categorize all functions into two: those employing proper cryptographic algorithms or assessing protected verification information and those that do not (see Section 4.3 for details). The next stage involves extracting function signatures, which consist of feature vectors and graph representations. The feature vector is derived through syntactic and structural analysis, while the graph representation is generated after constructing, normalizing, and pruning the DFG. Normalization serves to eliminate redundant nodes and edges, followed by pruning to remove elements irrelevant to the verification process. This approach is commonly used in existing research [47, 51, 78], ensuring accurate function matching across a wide array of possible implementations. Specifically, the process requires pinpointing the variables or arguments holding the essential verification data. After setting these as targets, program slicing is executed to maintain all relevant nodes and edges. For more detailed corpus statistics, please refer to Appendix A.3.

4.3 Vulnerability Discovery

Vulnerability Discovery Criteria. We focus on inspecting the implementation deviation of proper verification procedures for the four properties, i.e., authenticity, integrity, freshness, and compatibility. Despite the lack of verification accounts for most verification stage vulnerabilities, improper implementation of the verification procedure can also lead to an insecure or exploitable verification procedure. In our work, the primary concern of improper authenticity verification arises from the use of symmetric cryptographic algorithms (e.g., HMAC, CMAC, Poly1305). Likewise, we identify that the root cause of improper integrity verification often lies in the usage of weak digest verification algorithms such as CRC, SHA1, and MD5. Finally, for freshness and compatibility verification, our work emphasizes that the verification can be insecure when the verification information, including firmware version and compatible device ID, is extracted from unprotected data sources (e.g., filenames of firmware images).

Vulnerability Discovery Process. We identify vulnerabilities by examining both execution paths and associated verification procedures. If a firmware image contains only one execution path, we focus on that path. For firmware with multiple paths, we select the path with the most proper verification procedures, as this often indicates thorough verification. This strategy can reduce FPs, ensuring more conservative vulnerability identification. Next, vulnerability detection is performed on the sole or chosen path based on the previously defined criteria. Specifically, to identify missing verification vulnerabilities, we check for the absence of verification procedures for authenticity, integrity, freshness, or compatibility in the execution path. To detect improper verification vulnerabilities, we examine the use of improper functions in the corresponding verification procedures in the execution path. It is worth

Table 2: Firmware modification or selection for PoCs.

Type		Firmware Modification or Selection Method
Miss.	Auth.*	Select a firmware image without signature from the same vendor
	Integ.	Alter bits in an update firmware image
	Fresh.	Select an outdated firmware image for the same device
	Comp.*	Select an incompatible image from the same device family
Improp.	Auth.	Alter bits in an update image and resign it with the same key
	Integ.*	Alter bits in an update image and replace its digest
	Fresh.	Select an outdated image and alter its version field
	Comp.*	Select an incompatible image and alter its device ID field

Note: Miss.: Missing; Improp.: Improper; Auth.: Authenticity; Integ.: Integrity; Fresh.: Freshness; Comp.: Compatibility; Vulnerability types marked with an asterisk (*) most likely require patching during test environment generation.

noting that some firmware images perform a quick integrity check using a weak algorithm, followed by a more thorough verification for authenticity and integrity based on digital signature/MAC algorithms. We only report an alert for improper integrity verification when the digest algorithm used in the corresponding digital signature/MAC is also insecure.

4.4 Vulnerability Validation

PoC Creation. We dynamically validate an alert by feeding a PoC input that specifically violates a security property under test and observing whether the firmware update procedure still reaches the reboot stage. If it does, a vulnerability exists; if not, the alert is an FP. The PoC image varies based on alert type, as shown in Table 2. For example, to validate a missing compatibility verification, we replace the benign firmware with an incompatible version from the same device family. However, this process may also alter other properties, complicating root cause analysis. For example, the selected firmware image may also contain an incorrect version number. To mitigate the issue, ChkUp patch and repack the testing firmware to align the execution path of the verification procedure with the property under test.

Patch Generation. We patch to skip verification procedures that may hinder the vulnerability validation of a specific property under test. The patching is conducted either at the source code or binary level, depending on the program type. Typically, we invert conditional expressions without adding extra code, essentially altering comparisons to their opposites (e.g., equal to not equal). Listing 4 in Appendix A.4 presents the disassembly and decompiled code of the compatibility verification procedure of a TP-Link firmware image. This procedure fetches the current product ID using the *getProductVer()* function and extracts the compatible product ID of the new firmware from its file body. The ID comparison is conducted by a *bne* (branch if not equal) instruction. To bypass this verification with the PoC firmware, we can substitute *bne* with a *beq* instruction. Although most verification procedures employ straightforward conditional expressions, some include complex conditions involving various logical operators and multiple variables, some of which are only known at runtime. This complexity makes simple methods like negating

comparisons ineffective. To address this, we first conduct a successful firmware update with a benign image and record the values of variables involved in conditional expressions. Next, we perform static value-flow analysis, tracing backward from these variables to identify the instructions where their values are defined. Our investigation indicates that such instructions are typically within the same function, obviating the need for complex control-flow recovery [66]. Finally, with the recorded values and identified instructions, we employ in-place binary rewriting techniques [43] to make use of the previously recorded values, while including conditionals to ensure that the replacement applies only to the firmware execution phase of interest.

5 Implementation

ChkUp is prototyped to support both Linux-based and other embedded OS-based firmware images (equipped with file systems) across various architectures, including ARM, MIPS, and PowerPC. Specifically, the *Execution Path Recovery* module efficiently constructs a UFG for each firmware image within a 300s timeout, representing each UFG as a di-graph using NetworkX [13]. The control flow of various programs is analyzed with tools [22,61,66] like angr [66] for binaries. IPC and program invocation paradigms are identified based on the CPF module from KARONTE [59], extended to support both JavaScript and shell scripts. After constructing UFGs, execution path recovery is conducted using the Simple Paths module of NetworkX at the function level. In the *Verification Procedure Recognition* module, numerical features are extracted from source code (specifically for JavaScript and shell scripts), CFG, and both disassembled and decompiled code (leveraging Ghidra [14]). Upon constructing and normalizing DFGs using the normalization rules by Meijer et al. [51], Ullmann's algorithm is employed for DFG subgraph isomorphism to identify verification procedures. The *Vulnerability Discovery* module is based on the previous two modules and performs vulnerability discovery using the described criteria and process. In the *Vulnerability Validation* module, Firmadyne [18] is initially used for dynamic analysis, and if emulation is unsuccessful, the more advanced FirmAE [42] is employed. Additionally, Ghidra and Firmware-mod-kit [11] are used to patch and repack firmware images.

6 Evaluation

In this section, we evaluated the effectiveness three key modules of ChkUp, i.e., the *Execution Path Recovery* (Section 6.1), *Verification Procedure Recognition* (Section 6.2), and *Vulnerability Validation* (Section 6.3).

Datasets. We collected 157,141 firmware images from the websites of IoT vendors and successfully unpacked 111,958 of them. To enable large-scale analysis, a dataset, D_L, was

(a) Accuracy of update entry identification   (b) Metrics of procedure recognition   (c) Performance overhead
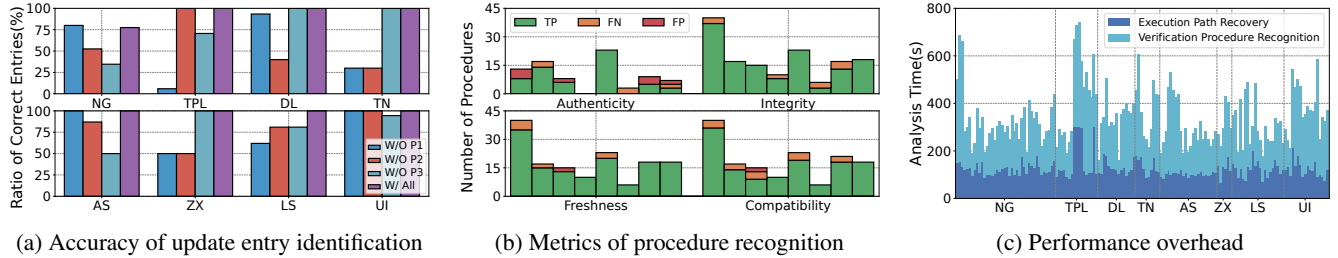
Figure 4: Evaluation results (NG, TPL, DL, TN, AS, ZX, LS, and UI represent Netgear, TP-Link, D-Link, TRENDnet, Asus, Zyxel, Linksys, and Ubiquiti, respectively).

created by randomly sampling 12,000 firmware images from eight major vendors, including Netgear, TP-Link, D-Link, TRENDnet, Asus, Ubiquiti, Zyxel, and Linksys. To evaluate the effectiveness of ChkUp and validate its alerts, a ground-truth dataset, D_G, was created by sampling from D_L. The construction of the ground truth for the firmware images in D_G was undertaken by four security experts through manual analysis. For details on the dataset construction and manual analysis, refer to Appendix B.1 and Appendix B.2.

Experimental Environment. The evaluation was conducted on a server with an Ubuntu 18.04 LTS OS and an AMD EPYC 7302P CPU with 64GB of RAM.

6.1 Effectiveness of Execution Path Recovery

The effectiveness of the *Execution Path Recovery* module is assessed by the accuracy of the update entry finding as well as the correctness of recovered execution paths.

Accuracy of Update Entry Finding. An update entry program is identified using three types of patterns: prompt message (P1), variable and function name (P2), and common code (P3). To evaluate the effectiveness of each type of pattern, we performed an ablation study, assessing the correctness of the identified entry programs under different settings, specifically, without P1, without P2, without P3, and with all patterns. Evaluation results for firmware from each vendor in D_G are shown in Figure 4a. The highest accuracy is achieved when all patterns are used, and the lowest when only P1 and P3 are employed. This indicates that P2 is the most essential for this process, as its absence often leads to significant performance drops across firmware images from most vendors. The impact of P1 and P3 varies by vendor. For instance, P1 significantly impacts firmware images from TP-Link, TRENDnet, and Zyxel, while P3 is vital for those from Netgear and Asus. Importantly, when using all patterns, the identification demonstrates its robustness by accurately identifying update entry programs of most firmware images. While 9 Netgear firmware images had their update entries misidentified due to limited semantic information, most of these misidentified programs are still related to firmware updates but handle update roles other than firmware delivery.

Correctness of Execution Path Recovery. Upon executing

the *Execution Path Recovery* module on firmware images from D_G, ChkUp takes an average of 126.0 seconds for each image (see Figure 4c). As a result, of the 150 generated UFGs, 122 are both firmware update sound and complete. 136 UFGs are sound since since every edge in UFGs represents control flows or IPC paradigms in the update procedures. However, UFGs from 7 Asus, 4 D-Link, and 3 Zyxel firmware images are unsound, yielding unrelated IPC paradigms. 133 UFGs are complete, containing all related control flows and IPC paradigms. However, UFGs from 9 Netgear firmware images are incomplete due to misidentified update entries. Note that the 3 above-mentioned UFGs from Zyxel firmware images are also incomplete due to a mismatch between update entries and back-end handlers. The rest, from 5 TP-Link firmware images, are incomplete due to timeouts during UFG construction.

We found that sound and complete UFGs always lead to the recovery of correct paths, while unsound or incomplete UFGs might introduce incorrect paths or overlook the correct ones during backward slicing. For example, the 3 unsound and incomplete UFGs of Zyxel firmware only contain reboot function invocations that were intended for other device management functionalities (e.g., applying new configurations). Despite this, most incorrect paths do not influence the vulnerability discovery process, as they contain relatively incomplete verification procedures and are filtered out as long as correct paths are also found during the vulnerability discovery. Also, all the overlooked correct paths still contain a reboot step and can be identified once the complete UFGs are constructed.

6.2 Effectiveness of Procedure Recognition

Upon evaluating the *Verification Procedure Recognition* module on D_G, ChkUp recognizes verification procedures for each firmware image in an average of 216.1 seconds (see Figure 4c). The results of recognizing different categories of verification procedures are shown in Figure 4b. Note that the eight columns in each category represent the results of firmware images from Netgear, TP-Link, D-Link, TRENDnet, Asus, Zyxel, Linksys, and Ubiquiti, respectively. In summary, there are 461 true positives (TPs), 45 false negatives (FNs), and 17 FPs. Fewer authenticity verification procedures are recognized because some execution paths indeed lack a firmware authenticity verification procedure, based on our analysis.

ChkUp has the highest integrity verification accuracy, as vendors often use standard functions such as *MD5_Update* with minor or even no customization.

False Result Discussion. The FPs and FNs primarily arise from inaccurate execution paths and misidentification of key functions. Specifically, our analysis reveals that inaccurate execution paths lead to 23 FNs and 6 FPs since either verification procedures are not included or unrelated code is mistakenly identified as verification procedures. For instance, device management functionalities, different from firmware updates, are included in the UFGs of 2 D-Link DAP firmware images and are falsely recognized as authenticity verification procedures. Other FNs and FPs mainly stem from using uncommon key functions or from including functions with similar semantics in execution paths. Notably, 6 FNs arise from deviating from the heuristics guiding the identification of verification procedures: 3 for Zyxel NBG-series images because return variables and arguments of their digest calculation functions do not feed into conditional expressions. A similar issue is seen in the version parsing of Netgear R-series images, resulting in 3 FNs. Further analysis shows that these images display the digests or parsed firmware versions on user interfaces, requiring manual verification. Despite some FNs and FPs, the module remains effective in most cases.

6.3 Effectiveness of Vulnerability Validation

To evaluate the effectiveness of the *Vulnerability Validation module*, we assessed its success rate on D_G and analyzed firmware patching results. Moreover, we measured its scalability on 1,200 additional firmware images from D_L.

Success Rate of Vulnerability Validation. After running the *Vulnerability Discovery* module on D_G, a total of 271 alerts were raised. Of these, 119 alerts were raised for 72 emulatable firmware images in D_G, which are compatible with the *Vulnerability Validation* module. Among the 119 alerts, 90 require patching to create a testing environment for conducting PoCs. Applying the *Vulnerability Validation* module on the corresponding firmware images resulted in 69 successful generations of patched firmware images. Obstacles in creating patched firmware images stem mainly from diverse firmware implementations, including the use of uncommon file systems, which are not supported by the state-of-the-art firmware repacking tool. We then emulated these repacked firmware images and found only 10 failed to run due to violations of runtime firmware signature checks. In total, 88 testing environments from both patched and original firmware images were created successfully. After undertaking PoC creation, 73 PoCs were successfully conducted and the corresponding alerts are considered TPs. Investigation of the failure cases reveals that 6 are indeed FPs and 9 are supposed to be TPs.

Patching Result Analysis. Unsuccessful PoC generation can result from either the alert being an FP or issues encountered during firmware patching. Specifically, patch generation in ChkUp involves three steps: 1) identification of verification procedures to bypass, 2) selection of code segments to modify, and 3) deployment of patched firmware. If the identification of the first step is incorrect, the subsequent patch might not enable further exploration of the program space, leading to an FN. Of the 15 failed PoC cases, 9 were due to this problem. Although not implemented in our current prototype, this issue can be mitigated by monitoring program execution to differentiate between known correct and incorrect running firmware. Even with the correct verification procedure identified, issues like heuristically negating logic in an incorrect code location can arise. Similar to the issue of misidentification, this can be addressed. Yet, we did not encounter any of these cases, likely due to the use of heuristics that work well for known firmware types. Lastly, the deployment of the patched firmware is not always feasible due to challenges with firmware repacking and emulation. A deeper analysis of these challenges is provided later.

Scalability of Vulnerability Validation. The scalability of the *Vulnerability Validation* module is closely tied to its ability to emulate and repack firmware images. To assess this, we analyzed a random sample of 1,200 firmware images, which represent 10% of D_L. Initial emulation tests showed that 44.1% of these images were successfully emulated and thus applicable to this module. Notably, D-Link firmware exhibited the highest emulation success rate, while Ubiquiti firmware had the lowest. For the emulatable images, we achieved a repacking success rate of 72.2%. The most significant factor hindering successful repacking was the presence of file systems that were not as widely supported as common ones like SquashFS and CramFS. Finally, of the repacked firmware images, 82.7% were successfully emulated. The primary reason for most failures was runtime firmware signature checks.

7 Vulnerability Discovery Results

7.1 Alerts on Real-world Firmware

Alert Analysis. We used ChkUp to identify potential vulnerabilities on D_L and analyzed the alert distribution. In terms of performance, ChkUp processed 93.4% of firmware images within 600 seconds each. Timeouts during UFG construction contributed to the extended analysis time observed in 3.4% of the firmware images. ChkUp resolved the execution paths for 10,670 firmware images and generated 15,132 alerts. Figure 5 displays the distribution of alerts by vulnerable verification type for major device types in D_L. Notably, a significant portion of alerts arise from firmware of various network devices (e.g., routers and switches) and cameras due to two primary reasons: 1) Network devices and cameras, which often have publicly accessible firmware images, dominate a substantial share of the IoT market; 2) Many devices reuse vulnerable

Table 3: Vulnerability discovery results of ChkUp on D_G dataset.

Vendor	# FW	Authenticity Verification						Integrity Verification						Freshness Verification						Compatibility Verification					
		Missing			*Improper*			*Missing*			*Improper*			*Missing*			*Improper*			*Missing*			*Improper*		
		TP	FN	FP	TP	FN	FP	TP	FN	FP	TP	FN	FP	TP	FN	FP	TP	FN	FP	TP	FN	FP	TP	FN	FP
Netgear	40	27	5	0	0	0	0	0	0	3	29	3	0	0	0	4	6	0	1	0	0	3	0	0	1
TP-Link	17	0	0	3	10	3	0	0	0	0	17	0	0	0	0	2	0	0	0	0	0	3	0	0	0
D-Link	15	9	2	0	0	0	0	0	0	0	9	0	0	0	2	0	9	0	0	0	2	4	5	4	0
TRENDnet	10	10	0	0	0	0	0	0	0	2	8	2	0	0	0	0	0	0	0	0	0	0	0	0	0
Asus	23	0	0	0	13*	0	0	0	0	0	0	0	0	0	0	0	4*	0	3	0	0	0	4*	0	4
Zyxel	6	3	0	3	0	0	0	0	0	3	3	3	0	0	0	0	0	0	0	0	0	0	0	0	0
Linksys	21	12	4	0	0	0	0	0	0	2	13	2	2	3	0	0	0	0	0	0	0	3	0	0	0
Ubiquiti	18	11	2	2	0	0	0	0	0	0	16	0	0	0	0	0	0	0	0	0	0	2	0	0	0
Summary	150	72	13	8	23	3	0	0	0	10	95	10	2	3	2	6	19	0	4	0	2	15	9	4	5

Note: Numbers marked with an asterisk (*) indicate that the TPs correspond to n-day vulnerabilities.
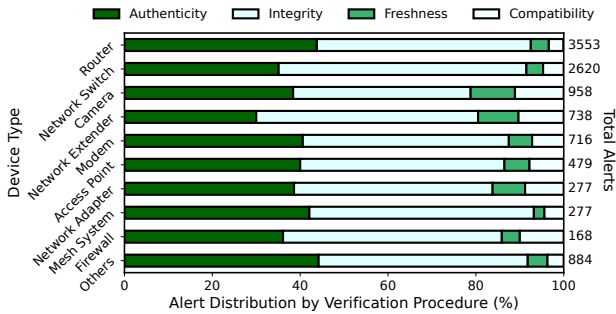

Figure 5: Distribution of alerts for various device types.

code for their firmware foundation and have seen a lack of effective security enhancements, resulting in persistent vulnerabilities across both new and old versions.

The most prevalent security issue reported by ChkUp involved vulnerable integrity verification procedures. For instance, of the alerts for network switch firmware, over half relate to this issue. The majority of these alerts highlight the use of weak algorithms like CRC and MD5. Vulnerable authenticity verification also threatens various device types, notably routers. The primary concern is the lack of verification, but the use of symmetric digital signatures is also widespread. While alerts related to freshness and compatibility verification are less common, cameras and network extenders exhibit more of these alerts than other devices. The most frequent issue involves using unprotected data, such as filenames, to verify versions or device IDs in the firmware update interfaces.

Invalid and Outlier Result Discussion. In the results from ChkUp, the invalid cases primarily consisted of 1,330 firmware images for which ChkUp could not resolve the execution paths. The outlier cases were those 589 firmware images that raised four alerts, which was uncommon as most images typically triggered up to three alerts. We delved deeper into these results by examining a random 10% sample. We found that two primary reasons emerged for the 133 failure cases in execution path recovery: 32.3% of cases stemmed from misidentification of the entry program, and 29.3% from timeouts during UFG construction. The remaining issues were mainly due to unusual IPC implementations or unsupported

programming languages, like PHP used for specific firmware update steps. We found no cases where, despite correct UFG construction, the use of reboot functions as slicing targets caused path identification to fail. Moreover, we inspected the verification procedures of 59 firmware images with four alerts to check their validity. Out of these, 39.0% were found to have insecure update mechanisms, while the remaining firmware contained FPs that primarily resulted from misidentified execution paths and verification procedures. Importantly, the key reason for the misidentification of verification procedures was the mismatch of functions between firmware images and the corpus. While most of the procedures that ChkUp overlooked do adhere to the value-to-condition heuristic, some do not. For these outliers that we confirmed as genuine, we provided exceptions, exempting them from the heuristic matching.

7.2 Real-world Vulnerabilities

Validating alerts is both time-consuming and labor-intensive, so our focus was on 150 firmware images from D_G. We employed the *Vulnerability Validation* module for emulatable firmware and manually validated the rest with our ground truth. To pinpoint FNs, all firmware images were assessed using the ground truth. The results are displayed in Table 3. Overall, the TPR is 86.7% and the FPR is 5.3%, demonstrating the effectiveness of ChkUp in detecting vulnerabilities. Most FPs and FNs arose from the *Execution Path Recovery* and *Verification Procedure Recognition* modules as we discussed in our evaluation. TPs for Asus firmware images are n-day vulnerabilities (i.e., CVE-2014-2718, CVE-2020-15498, and CVE-2021-3166), while those for 29 other device families were undisclosed. We have also reported our findings to vendors and received acknowledgment. At the time of writing, 25 CVE IDs and one PSV ID have been assigned. It is worth noting that the majority fall into the categories of CWE-345 and CWE-20. These findings align with our systematization, indicating that these two categories are the most prevalent. More CVE/PSV details are available at our project website [76].

Vulnerability Analysis. Regarding the vulnerability type, improper integrity verification emerged as the most frequent issue, with 39.0% using CRC and 23.8% using MD5. Issues

```
1. upgrade.js
01.function clickUpgrade(form){...
02.  var file_array=file_name.split('-V');...
03.  // check the firmware compatibility with filename
04.  var file_module=file_array[0];
05.  if(file_module.toUpperCase()!=netgear_module.toUpperCase()){
06.    alert(error_module); return false;}...
07.  // check the firmware freshness with filename
08.  if(netgear_num>file_num){
09.    if(!confirm(oldver1+file_version+oldver2+netgear_version+oldver3))
10.      return false;}... return true;}        Front-end
--------------------------------------------------------------
2. webupgrade.sh                                  Back-end
01.imginstall(){...
02.  # check the firmware compatibility with firmware header
03.  module_name=$(cat /module_name)
04.  new_name=$(sed -n '1{p; q}' $INFO_HEAD | sed 's/.*://')
05.  if [ "$module_name" != "$new_name" ]; then...
06.    giveup_webupgrade_in_imginstall $STATUS_CHKINFO_ERROR fi
07.  # check the firmware integrity using CRC checksum
08.  CHECKSUM=$(/sbin/mychecksum -o $offset -i $IMPORT_FILE_NAME|sed
's/,.*$//')
09.  if [ "$CHECKSUM" != "checksum = 0x00" ]; then...
10.    giveup_webupgrade_in_imginstall $STATUS_CHKSUM_ERROR... fi...}

3. mychecksum (decompiled code)
01.bool calcsum(undefined4 param_1,__off_t param_2){...
02.  while (sVar3 = read(__fd,local_a8,0x80), 0 < sVar3){...
03.    // calculate the CRC checksum
04.    iVar6 = iVar6 + sVar3;}
05.  printf("checksum = 0x%02X, len = %d\n",~uVar5 & 0xff,iVar6);...}
```

Listing 3: Firmware update code in the case studies.

with missing and improper authenticity verification are prevalent. For example, in the firmware from TP-Link WR-series devices, although there exists an RSA signature for authenticity verification in firmware updates, the signature does not protect the firmware header. Moreover, the MD5 sum value is also stored in the header. Therefore, to craft a malicious firmware image that can bypass the verification, an attacker could modify header content and recompute the hash value to replace the original one. Furthermore, fewer vulnerabilities were identified in freshness and compatibility, mostly rooted in the use of unprotected information. For instance, the firmware version verification of D-Link DAP-series devices is to extract the version from the filename of the new firmware image and compare it with the current version. As the filename lacks authenticity and integrity protections, attackers could alter the filename to bypass the verification and introduce vulnerable firmware images to devices.

7.3 Case Studies

Listing 3 shows the firmware verification flows in a firmware image from Netgear WNR-series routers. These devices provide a web interface for manual firmware updates. In the front end of the interface, as seen in line 3 to line 10 of *upgrade.js*, both compatibility and freshness are verified by examining the filename of the uploaded file, a method we have already identified as vulnerable. After passing these verifications, the back-end shell script *webupgrade.sh* proceeds with further verification. From line 2 to line 6 of *webupgrade.sh*, the firmware header is examined to ensure compatibility. Therefore, compatibility can be ensured as long as firmware integrity is ensured. Integrity verification occurs in *webupgrade.sh* (see line 7 to line 10) through the binary *mychecksum*. Upon inspection, *mychecksum* uses the weak CRC algorithm for this verification. Notably, authenticity verification is absent. As a result, these vulnerable verification procedures make the firmware

update mechanism susceptible to real-world exploits.

Real-world Exploits. We showcase the exploitability of these vulnerabilities by crafting two exploits, specifically a *firmware downgrade attack* and a *firmware modification attack*, targeting a Netgear WNR-series router. In these attack scenarios, both attackers and the target device share the same network environment. Attackers can either directly access the firmware update interface or sniff the network and initiate MITM attacks, given that the communication for firmware delivery uses unencrypted HTTP. Notably, during our preliminary manual firmware analysis, we found a significant number of public firmware images lack TLS for update interfaces, highlighting the feasibility of these attacks in the real world.

A1. Firmware Downgrade Attack: Since the firmware freshness is determined by checking the filename of the uploaded file, attackers can craft a malicious firmware image using a legacy firmware image by changing its version field in the filename to match a legitimate one. Then, they can either upload the malicious firmware image through the web interface or substitute the benign firmware image through MITM. Subsequently, the firmware is replaced with an insecure legacy version with vulnerabilities, which can be further exploited.

A2. Firmware Modification Attack: As the firmware integrity verification is based on CRC checksum, attackers can create a modified firmware image while keeping the checksum value consistent. However, some fields, such as device ID in the header for compatibility verification, need to remain unchanged to ensure other verification procedures proceed successfully. Attackers can introduce the malicious firmware to the device in the same way mentioned in *firmware downgrade attack*. With this attack, if the malicious firmware is carefully crafted, various further attacks can be introduced, including backdoors, malware, and DoS attacks.

8 Discussion

Security Impacts of Heuristics. Heuristic approaches capture common patterns discovered via manual reverse engineering. While effective for firmware images with similar patterns, they cannot capture the foundational problem that can be generalized across different systems. In the context of ChkUp, our approach that attempts to heuristically identify the entry and end of firmware update paths can fail on customized firmware, preventing the analysis from starting. Furthermore, our heuristic approach that uses information flow from the crypto function to identify the update phase could fail to extract the correct update phase code. From the evaluation, we found that such heuristics can fail on unconventional designs.

Limitations of Static and Dynamic Analysis. There is a trade-off between static analysis and dynamic analysis. However, when used appropriately in combination, they provide a good balance between completeness and soundness of analysis. In the context of ChkUp, a key advantage of static analysis

is the elimination of the need to emulate firmware, which remains an open research challenge. For example, out of 150 firmware images in D_G, only 72 are emulatable using state-of-the-art firmware emulators. The emulation rate is even lower for bare-metal firmware. Another advantage of static analysis is the ability to detect deeper bugs where crafting an input to reach the bug is extremely challenging. For example, 39 firmware images check the cryptographic signature before proceeding to the buggy verification procedures. Yet, static analysis often results in many false alarms. To mitigate this, ChkUp leverages dynamic analysis to attempt to provide a level of validation, albeit a basic one. Besides the challenge of emulation, dynamic analysis only allows the validation of one path at a time, limiting the breadth of exploration. For ChkUp, success also depends on the patching being performed correctly, which is not always true as discussed.

Extensibility of ChkUp. Beyond firmware update vulnerabilities, our techniques can detect vulnerabilities like faulty firmware function implementations (e.g., in cryptographic functions). By adding faulty function signatures to the corpus, we ensure that even if there are correctly implemented counterparts, the vulnerable function will match its signature based on a higher similarity score, enabling us to pinpoint the use of flawed implementations. ChkUp is tailored for multi-architecture Linux and other embedded OS firmware with file systems. While bare-metal firmware analysis poses challenges, like addressing base resolution, most do not require cross-language control/data flow analysis. By incorporating methods from FirmXRay [77] and updating the corpus, ChkUp could handle bare-metal firmware too. Our technique is not vendor-specific; if ChkUp supports the firmware type, it can analyze it. Though FPs can occur, our system can still offer insights like firmware update paths. Extending the corpus can further improve the accuracy of ChkUp.

9 Related Work

Firmware Update Security. Recent studies have revealed security concerns in firmware or software update mechanisms [15, 17, 21, 57, 63, 70]. Notably, the prevalent use of insecure protocols like HTTP can expose update processes to MITM and backdoor attacks [17, 63]. Moreover, there are demonstrated firmware modification attacks by exploiting update procedure weaknesses [21, 70]. Both academics and the Internet Engineering Task Force (IETF) are addressing these concerns by developing secure firmware update strategies [46, 53] and efficient hotpatching solutions [35, 54].

Vulnerability Detection in Firmware. Firmware vulnerability detection is broadly divided into three categories. The first group [37, 40, 68, 77, 82, 87] detects vulnerabilities by identifying discrepancies between specifications and actual implementations. For example, FirmXRay [77] uncovers Bluetooth layer vulnerabilities using specification knowl-

edge. The second category uses methods like taint analysis [19, 27, 59, 83, 85] and symbolic execution [25, 32, 34, 36, 65] to explore vulnerable execution paths. For instance, Karonte [59] targets memory-corruption vulnerabilities by pinpointing unsafe user-controlled input sinks. The third category involves pattern matching [20, 38, 44, 81] and code similarity checks [23, 24, 28, 33, 39, 41, 49, 50, 64, 75, 79, 88], detecting vulnerabilities by matching known patterns or vulnerable code, such as FirmUP [24] that uses procedure similarity. Our research specifically targets firmware update vulnerabilities, which are distinct due to their multi-stage nature and the unique semantics of firmware updates.

Cryptographic Misuse Detection: Cryptographic API misuse detection is a technique for identifying potential vulnerabilities stemming from incorrect usage of cryptographic primitives [16]. The general idea is to verify that parameters passed to cryptographic APIs meet pre-defined rules [26, 29, 31, 45, 56, 58, 67, 85]. ChkUp differs from existing cryptographic misuse detection studies in terms of system goals, security impact and technical standpoints. Regarding security goals, ChkUp focuses on firmware update security, providing a fundamentally distinct focus. Instead of analyzing individual cryptographic vulnerabilities, our approach involves systematizing the firmware update ecosystem and conducting a security analysis to map the attack surface, uncovering a significant amount of non-crypto attack vectors. From the perspective of security impact, ChkUp provides new ways to automate the search for firmware update vulnerabilities, covering both crypto misuses and non-crypto-related vulnerabilities such as downgrade attacks. From the technical perspective, ChkUp tackles new challenges, including identifying long sequences of inter-process invocations as well as discovering and validating firmware update-specific semantic bugs.

10 Conclusion

In this paper, we present ChkUp, a novel approach for detecting firmware update vulnerabilities, including missing and improper verification during updates. Specifically, ChkUp resolves firmware update execution paths through cross-language inter-process control flow analysis and program slicing. Then, firmware verification procedures are identified through syntactic, structural, and semantic program analysis. These procedures along with the corresponding execution paths are further examined based on our defined criteria to detect vulnerabilities. To reduce false positives, alerts for emulatable firmware images are validated dynamically with a patching-based method, while others are validated manually. ChkUp is implemented and employed to analyze 12,000 firmware images, with subsequent validation of alerts for 150 firmware images from 33 device families. The results show that ChkUp can identify zero-day and n-day vulnerabilities, leading to the assignment of 25 CVE IDs and one PSV ID.

References

[1] LibCRC – Open Source CRC Library in C. https://www.libcrc.org/.

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

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

[4] Mbed Crypto. https://os.mbed.com/docs/mbed-os/v6.16/apis/mbed-crypto.html.

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

[6] wolfSSL. https://www.wolfssl.com/doxygen/.

[7] zlib. https://www.zlib.net/.

[8] CVE-2018-3926. https://nvd.nist.gov/vuln/detail/CVE-2018-3926, 2018.

[9] CVE-2021-3166. https://nvd.nist.gov/vuln/detail/CVE-2021-3166, 2021.

[10] Embedded devices market size report. https://www.marketsandmarkets.com/Market-Reports/embedded-system-market-98154672.html, 2022.

[11] Firmware-mod-kit. https://github.com/rampageX/firmware-mod-kit, 2022.

[12] Internet of things (iot) top 10 2018. https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Top_10, 2022.

[13] Network analysis in python. https://github.com/networkx/networkx, 2022.

[14] National Security Agency. Ghidra software reverse engineering framework. https://github.com/NationalSecurityAgency/ghidra, 2022.

[15] Omar Alrawi et al. Sok: Security evaluation of home-based iot deployments. In *S&P*, 2019.

[16] Amit Seal Ami et al. Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In *S&P*, 2022.

[17] Anthony Bellissimo et al. Secure software updates: Disappointments and new challenges. In *HotSec*, 2006.

[18] Daming D Chen et al. Towards automated dynamic analysis for linux-based embedded firmware. In *NDSS*, 2016.

[19] Libo Chen et al. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *USENIX Security*, 2021.

[20] Andrei Costin et al. A {Large-Scale} analysis of the security of embedded firmwares. In *USENIX Security*, 2014.

[21] Ang Cui et al. When firmware modifications attack: A case study of embedded exploitation. *NDSS*, 2013.

[22] Piotr Dabkowski. Js2py: Javascript to python translator. https://github.com/PiotrDabkowski/Js2Py, 2022.

[23] Yaniv David et al. Similarity of binaries through re-optimization. In *PLDI*, 2017.

[24] Yaniv David et al. Firmup: Precise static detection of common vulnerabilities in firmware. *ASPLOS*, 2018.

[25] Drew Davidson et al. {FIE} on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security*, 2013.

[26] Manuel Egele et al. An empirical study of cryptographic misuse in android applications. In *CCS*, 2013.

[27] Mohamed Elsabagh et al. {FIRMSCOPE}: Automatic uncovering of {Privilege-Escalation} vulnerabilities in {Pre-Installed} apps in android firmware. In *USENIX Security*, 2020.

[28] Sebastian Eschweiler et al. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

[29] Sascha Fahl et al. Why eve and mallory love android: An analysis of android ssl (in) security. In *CCS*, 2012.

[30] Andrew Fasano et al. Sok: Enabling security analyses of embedded systems via rehosting. In *ASIACCS*, 2021.

[31] Johannes Feichtner et al. Automated binary analysis on ios: a case study on cryptographic misuse in ios applications. In *WiSec*, 2018.

[32] Farhaan Fowze et al. Proxray: Protocol model learning and guided firmware analysis. *IEEE Trans. Softw. Eng.*, 2019.

[33] Jian Gao et al. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *ASE*, 2018.

[34] Fabio Gritti et al. Heapster: Analyzing the security of dynamic allocators for monolithic firmware images. In *S&P*, 2022.

[35] Yi He et al. Rapidpatch: Firmware hotpatching for real-time embedded devices. In *USENIX Security*, 2022.

[36] Grant Hernandez et al. Firmusb: Vetting usb device firmware using domain informed symbolic execution. In *CCS*, 2017.

[37] Grant Hernandez et al. {BigMAC}:{Fine-Grained} policy analysis of android firmware. In *USENIX Security*, 2020.

[38] Grant Hernandez et al. Firmwire: Transparent dynamic analysis for cellular baseband firmware. *NDSS*, 2022.

[39] Dongkwan Kim et al. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Trans. Softw. Eng.*, 2022.

[40] Eunsoo Kim et al. Basespec: Comparative analysis of baseband software and cellular specifications for l3 protocols. In *NDSS*, 2021.

[41] Geunwoo Kim et al. Improving cross-platform binary analysis using representation learning via graph alignment. In *ISSTA*, 2022.

[42] Mingeun Kim et al. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *ACSAC*, 2020.

[43] Taegyu Kim et al. Revarm: A platform-agnostic arm binary rewriter for security applications. In *ACSAC*, 2017.

[44] Taegyu Kim et al. {PASAN}: Detecting peripheral access concurrency bugs within {Bare-Metal} embedded applications. In *USENIX Security*, 2021.

[45] Stefan Krüger et al. Crysl: An extensible approach to validating the correct usage of cryptographic apis. *IEEE Trans. Softw. Eng.*, 2019.

[46] Antonio Langiu et al. Upkit: An open-source, portable, and lightweight update framework for constrained iot devices. In *ICDCS*, 2019.

[47] Pierre Lestringant et al. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *ASIACCS*, 2015.

[48] Wen Li et al. Understanding language selection in multi-language software projects on github. In *ICSE-Companion*, 2021.

[49] Bingchang Liu et al. αdiff: cross-version binary code similarity detection with dnn. In *ASE*, 2018.

[50] Andrea Marcelli et al. How machine learning is solving the binary function similarity problem. In *USENIX Security*, 2022.

[51] Carlo Meijer et al. Where's crypto?: Automated identification and classification of proprietary cryptographic primitives in binary code. In *USENIX Security*, 2021.

[52] Charlie Miller et al. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

[53] Brendan Moran et al. A firmware update architecture for internet of things. *Internet Requests for Comments, RFC Editor, RFC 9019*, 2021.

[54] Christian Niesler et al. Hera: Hotpatching of embedded real-time applications. In *NDSS*, 2021.

[55] U.S. Government Accountability Office. Solarwinds cyberattack demands significant federal and private-sector response (infographic). https://www.gao.gov/blog/solarwinds-cyberattack-demands\-significant-federal-and-private-sector-\response-infographic, 2021.

[56] Luca Piccolboni et al. Crylogger: Detecting crypto misuses dynamically. In *S&P*, 2021.

[57] Vijay Prakash et al. Inferring software update practices on smart home iot devices through user agent analysis. In *SCORED*, 2022.

[58] Sazzadur Rahaman et al. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *CCS*, 2019.

[59] Nilo Redini et al. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *S&P*, 2020.

[60] ReFirmLabs. binwalk. https://github.com/ReFirmLabs/binwalk/, 2022.

[61] Yann Régis-Gianas et al. Morbig: A static parser for posix shell. In *SLE*, 2018.

[62] Michael Rushanan et al. Sok: Security and privacy in implantable medical devices and body area networks. In *S&P*, 2014.

[63] Justin Samuel et al. Survivable key compromise in software update systems. In *CCS*, 2010.

[64] Paria Shirani et al. Binarm: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In *DIMVA*, 2018.

[65] Yan Shoshitaishvili et al. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[66] Yan Shoshitaishvili et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *S&P*, 2016.

[67] David Sounthiraraj et al. Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps. In *NDSS*, 2014.

[68] Lin Tan et al. Autoises: Automatically inferring security specification and detecting violations. In *USENIX Security*, 2008.

[69] EE Times. 2019 embedded markets study. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, 2019.

[70] Ryan Tsang et al. Fandemic: Firmware attack construction and deployment on power management integrated circuit and impacts on iot applications. In *NDSS*, 2022.

[71] Julian R Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 1976.

[72] Jinwen Wang et al. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *S&P*, 2022.

[73] Jinwen Wang et al. Ari: Attestation of real-time mission execution integrity. In *USENIX Security*, 2023.

[74] Jinwen Wang et al. IP Protection in TinyML. In *DAC*, 2023.

[75] Shuai Wang et al. In-memory fuzzing for binary code similarity analysis. In *ASE*, 2017.

[76] [Website]. ChkUp. https://fw-chkup.github.io.

[77] Haohuang Wen et al. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *CCS*, 2020.

[78] Seongil Wi et al. Hiddencpg: large-scale vulnerable clone detection using subgraph isomorphism of code property graphs. In *WWW*, 2022.

[79] Yueming Wu et al. Detecting semantic code clones by building ast-based markov chains model. In *ASE*, 2022.

[80] Yuhao Wu et al. Work-in-progress: Measuring security protection in real-time embedded firmware. In *RTSS*, 2022.

[81] Fabian Yamaguchi et al. Modeling and discovering vulnerabilities with code property graphs. In *S&P*, 2014.

[82] Yuqing Yang et al. Detecting and measuring misconfigured manifests in android apps. In *CCS*, 2022.

[83] Jiawei Yin et al. Finding smm privilege-escalation vulnerabilities in uefi firmware with protocol-centric static analysis. In *S&P*, 2022.

[84] Zhiyuan Yu et al. Security and privacy in the emerging cyber-physical world: A survey. *IEEE Commun. Surv. Tutor.*, 2021.

[85] Li Zhang et al. {CryptoREX}: Large-scale analysis of cryptographic misuse in {IoT} devices. In *RAID*, 2019.

[86] Ruide Zhang et al. Augauth: Shoulder-surfing resistant authentication for augmented reality. In *ICC*, 2017.

[87] Yue Zhang et al. When good becomes evil: Tracking bluetooth low energy devices via allowlist-based side channel and its countermeasure. In *CCS*, 2022.

[88] Binbin Zhao et al. A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware. In *ISSTA*, 2022.

A Additional Design Details

A.1 Path Explosion Reduction

Five path explosion reduction strategies are utilized in ChkUp. The initial three are applied during UFGs construction, and the last two during backward slicing. 1) Excluding non-crypto standard libraries: Standard libraries are omitted from UFG construction to reduce complexity, with an exception for cryptographic libraries for vulnerability identification without significantly increasing UFG complexity. 2) Omitting built-in utilities: During firmware updates, known built-in utility programs (e.g., *mtd*, *reboot*) are executed, negating the need for control flow analysis. 3) Implementing a timeout: If UFG generation takes excessive time, a timeout strategy limits UFG complexity and the inclusion of FP paths. 4) Applying path filtering: Execution paths are refined by filtering based on error messages from unsuccessful firmware updates causing device reboots. 5) Path merging: Post backward slicing, paths with identical verification procedures and nodes are merged for their equivalent semantic meanings.

A.2 Function Similarity Matching

Similarity Score Calculation. With extracted features (see details in Table 4), we can calculate the similarity between a function f and a function in the corpus f' based on the relative difference between their feature values [39]. Given

Table 4: Syntactic and structural features of functions.

Category	Feature
Data Constant	# constants, # strings
Instruction	# all instructions, # operands, # each type of instructions[1]
CFG	# BBs, # edges, # loops, avg. # edges per BB, * BBs, * loops
Function Call	# imported calls, # incoming calls, # outgoing calls
Misc.	# arguments, # API callees, # library references, # code references

Note: #: The number of; *: The size of; avg.: average. [1] Instruction type: arithmetic, branch, data transfer, logic, and bit-oriented instructions.
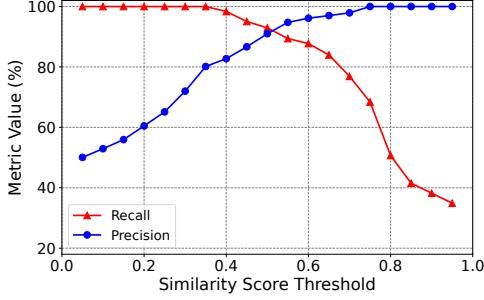
Figure 6: Effect of threshold selection on function matching.

M features, the feature vectors for f and f' are represented as $\mathbf{f} = [x_1, x_2, \cdots, x_M]$ and $\mathbf{f}' = [x_1', x_2', \cdots, x_M']$, respectively. The relative difference δ between x_i and x_i' is

$$\delta(x_i, x_i') = \frac{|x_i - x_i'|}{Max(|x_i|, |x_i'|)}. \tag{1}$$

From this, the similarity score between the two features is $1 - \delta(x_i, x_i')$. Then, the overall similarity score of f and f' is defined as the average similarity score for all features by

$$\gamma(\mathbf{f}, \mathbf{f}') = 1 - \frac{1}{M}(\delta(x_1, x_1') + \delta(x_2, x_2') + \cdots + \delta(x_M, x_M')). \tag{2}$$

The value ranges of $\delta(x_i, x_i')$ and $\gamma(\mathbf{f}, \mathbf{f}')$ are 0 to 1. The higher the $\gamma(\mathbf{f}, \mathbf{f}')$, the more similar f and f' are considered to be.

Based on this similarity score definition, we assess the similarity between each function in the execution paths and each function in the corpus. The corpus contains Q key functions divided into four sets: $\mathbf{S}_n^a$, $\mathbf{S}_n^i$, $\mathbf{S}_n^f$, and $\mathbf{S}_n^c$. Given N recovered execution paths of a firmware image, we calculate four similarity matrices for each path including $\mathbf{S}_n^a$, $\mathbf{S}_n^i$, $\mathbf{S}_n^f$, and $\mathbf{S}_n^c$. Each matrix represents the similarity scores between functions in the n-th path and those in the corpus for a specific verification procedure. For instance, the similarity score matrix $\mathbf{S}_n^a$ for authenticity verification of the n-th path is defined as

$$\mathbf{S}_n^a = \begin{bmatrix} \gamma(\mathbf{f}_1, \mathbf{f}_1') & \gamma(\mathbf{f}_1, \mathbf{f}_2') & \cdots & \gamma(\mathbf{f}_1, \mathbf{f}_Q') \\ \gamma(\mathbf{f}_2, \mathbf{f}_1') & \gamma(\mathbf{f}_2, \mathbf{f}_2') & \cdots & \gamma(\mathbf{f}_2, \mathbf{f}_Q') \\ \vdots & \vdots & \vdots & \vdots \\ \gamma(\mathbf{f}_P, \mathbf{f}_1') & \gamma(\mathbf{f}_P, \mathbf{f}_2') & \cdots & \gamma(\mathbf{f}_P, \mathbf{f}_Q') \end{bmatrix}, \tag{3}$$

where P is the number of functions in the execution path and $\gamma(\mathbf{f}_p, \mathbf{f}_q')$ is the similarity score between the p-th function in the execution path and the q-th function in the corpus, as calculated using Equation 2. Note that when there is function overlap in the execution paths, the similarity score between each function pair is only calculated once.

Similarity Score Threshold Selection. In the first stage of the *Verification Procedure Recognition*, the similarity score threshold should effectively eliminate a substantial number of irrelevant functions while preserving the majority of essential key functions. This approach can ensure efficient and accurate recognition of the verification procedure in the second stage. To establish this threshold, we calculated numerous similarity scores between key functions found in the firmware images from D_G and their counterparts in the corpus. Moreover, we

Table 5: Cryptographic functions in the corpus.

Library	Integrity		Authentication	
	Proper	*Improper*	*Proper*	*Improper*
Libcrypto [2]	SHA256_Update, SHA3_absorb, RIPEMD160_Update, etc.	MD4_Update, MD5_Update, SHA1_Update, etc.	RSA_verify, DSA_verify, ECDSA_do_verify, etc.	HMAC_Update, CMAC_Update, Poly1305_Update, etc.
wolfCrypt [6]	Sha256Update, Sha3_512_Update, RipeMdUpdate, etc.	Md4Update, Md5Update, ShaUpdate, etc.	RsaSSL_Verify, DsaVerify, ecc_verify_hash, etc.	HmacUpdate, CmacUpdate, Poly1305Update, etc.
LibTomCrypt [3]	sha256_process, sha3_process, rmd160_process, etc.	md4_process, md5_process, sha1_process, etc.	rsa_verify_hash, dsa_verify_hash, ecc_verify_hash, etc.	hmac_process, omac_process, poly1305_process, etc.
Mbed Crypto [4]	sha256_update_ret, sha512_process, ripemd160_update_ret, etc.	md4_update_ret, md5_update_ret, sha1_update_ret, etc.	rsa_pkcs1_verify, ecdsa_verify	cipher_cmac_update, md_hmac_update, poly1305_update, etc.
Nettle [5]	sha256_update, sha3_update, ripemd160_update, etc.	md4_update, md5_update, sha1_update, etc.	rsa_sha512_verify, dsa_verify, ecdsa_verify, etc.	hmac_sha1_update, cmac_aes128_update, poly1305_aes_update, etc.

Note: Function names in wolfCrypt, Mbed Crypto, and Nettle are prefixed with *wc_*, *mbedtls_*, and *nettle_*, respectively; Omitted functions implement algorithms from the same family of algorithms implemented by the listed functions, for example, MD2 for improper integrity verification and SHA512 for proper integrity verification.

selected an irrelevant function at random for each key function in every firmware image and calculated its similarity score with the corresponding function in our corpus. In total, we assessed 1,012 functions and generated their respective similarity scores. Subsequently, we defined a range of similarity score thresholds, ranging from 0 to 1 in increments of 0.05, and measured the recall and precision at each threshold. As illustrated in Figure 6, a threshold of approximately 0.5 delivers an optimal balance between precision and recall, with both metrics exceeding 90%. Consequently, we selected a similarity threshold of 0.5.

A.3 Corpus Statistics

Key functions in the corpus include the functions from open-source libraries and proprietary functions obtained during the construction of the ground truth. Overall, the corpus contains 129 functions: 76 from widely used libraries and 53 that are proprietary. Our observations on key functions used for integrity and authentication verification align with previous work [85], showing that executable programs generally utilize either low-level cryptographic APIs from standard libraries or employ self-defined APIs that wrap these low-level APIs. Therefore, our corpus includes common cryptographic functions for digest algorithms (such as SHA family, MD family, and RIPEMD family), digital signature algorithms (such as RSA, DSA, and ECDSA), and MAC algorithms (such as HMAC, CMAC, and Poly1305) from standard libraries, namely *Libcrypto* [2], *wolfCrypt* [6], *LibTomCrypt* [3], *Mbed Crypto* [4], *Nettle* [5]. These functions are further classified into two categories (proper and improper) based on their corresponding algorithms (see Table 5 for details). In addition, non-cryptographic digest functions based on CRC, considered weak for integrity verification, from *zlib* [7] and *LibCRC* [1] are included. All these functions are compiled for ARM (both 32-bit and 64-bit), MIPS (both 32-bit and 64-bit), and PowerPC (both 32-bit and 64-bit) architectures using the GCC compiler with optimization levels ranging from O0 to O3.

Proprietary functions in the corpus include those used for

```
Disassembly Code                       Decompiled Code
01.lw   s0,0x44(s2)                     01.iVar5 = *(int *)(param_1
02.jalr t9=>getProductVer               + 0x44);
                                        02.iVar1 = getProductVer();
-03.bne s0,v0,LAB_004e9c4c            -03.if (iVar5 != iVar1) {
+04.beq s0,v0,LAB_004e9c4c            +04.if (iVar5 == iVar1) {
05._li  v1,0x4655                       05.    return 0x4655;
06.clear v1                             06.}
07.LAB_004e9c4c:                        07.return 0;
08.lw   ra,local_4(sp)
09.move v0,v1
10.jr   ra
```

Listing 4: A compatibility verification patch example.

freshness and compatibility verification and developer-defined functions for integrity and/or authenticity that significantly differ from open-source alternatives. These functions were collected through our manual analysis while constructing the ground truth using firmware images from eight different vendors, as outlined in D_G. During this process, we observed that key functions in firmware images from the same device family or vendor tend to be similar. For example, all firmware images from the Asus DSL-N families in D_G feature similar key functions. As a result, the proprietary functions in the corpus can be scaled across numerous firmware images besides their source firmware images. To facilitate vulnerability discovery, all functions are classified into proper and improper categories, based on whether the information used for verification is protected as mentioned in Section 4.3.

A.4 Patching Example

Listing 4 shows the disassembled and decompiled code for the compatibility verification in a TP-Link firmware image. This procedure can be bypassed with our patching method.

B Firmware Collection and Manual Analysis

B.1 Evaluation Datasets

Firmware Collection and Unpacking. We developed a web crawler to collect firmware images primarily from official websites of major vendors in areas like network devices, cameras, and smart home devices [80]. The collected 157,141 firmware images from 204 vendors were then unpacked using Binwalk [60] to extract the file system, kernel, and bootloader, successfully unpacking 111,958 firmware images (statistics can be found in our website [76]). Given the differences in firmware update mechanisms across various vendors, building a ground truth for evaluating ChkUp with firmware images from all vendors demands substantial manual work. Thus, we built a large-scale dataset, D_L, by randomly sampling 12,000 (just over 10%) firmware images consisting of eight leading vendors (i.e., Netgear, TP-Link, D-Link, TRENDnet, Asus, Ubiquiti, Zyxel, and Linksys) from our collection of unpacked firmware images. The eight vendors have significant market share in the embedded device market, notably holding over 60% market share in the wireless router market.

Ground Truth Dataset. Thoroughly evaluating ChkUp and validating its alerts requires manual analysis to establish ground truth, a labor-intensive process even for experts. To establish a ground-truth dataset, D_G, we performed a weighted random sampling of 150 firmware images from D_L, prioritizing those from various device families. These selected images originate from 33 different device families of the eight leading vendors. Then, manual analysis is performed by four security research field experts to build ground truth for D_G (for details, refer to Appendix B.2). With the ground truth, D_G is used to evaluate the effectiveness of ChkUp in Section 6 and validate the generated alerts by ChkUp in Section 7.

B.2 Manual Firmware Analysis

Assumption. Since a ground truth dataset is theoretically impossible to obtain, we have to assume that our manual analysis and cross-validation among different members provide a good approximation to the ground truth.

Approach. A four-member team of experienced security researchers performed the analysis. One team member is a senior computer security researcher with almost 20 years of experience, while the other three members have about 7 years of exposure to computer security. Given a firmware image, the objective is to figure out its firmware update mechanism by resolving all the invoked programs and their execution sequences, as well as all the critical verification procedures. To this end, we use a multi-step, documentation-supported approach. First, we search all the programs suspected of containing firmware update functionalities based on firmware update-related keywords like *firmware update* and *firmware upgrade*. We perform a careful review of any available firmware documentation and README files. These resources are invaluable for validating hypotheses about the function of specific binaries, effectively compensating for the absence of embedded information. For instance, if the screenshots of update interfaces are provided in the documents, the corresponding front-end programs can be easily found.

Secondly, we conduct a control flow analysis to understand the program execution sequence using the binary analysis tool, Ghidra. Data flow analysis, facilitated by angr, reveals how data moves and changes within the programs, providing insights into critical verification procedures. For emulatable firmware images, we also perform emulation to record the firmware update execution paths. Finally, we perform cross-validation to mitigate the limitations in analysis accuracy stemming from the varying experiences of researchers. Specifically, team members independently assess the same binaries and subsequently reconcile their findings. This process depends on internal peer reviews and external documentation to arbitrate discrepancies. Consequently, our approach guarantees a comprehensive examination, establishing a robust foundation for a reliable ground truth.