Sensors and Materials, Vol. 38, No. 7 (2026) 4027–4048
MYU Tokyo

S & M 4548

Improvement for Embedded Firmware Emulation Applying FirmAE

Jong-Yih Kuo,* Yu-Quan Wang, Ti-Feng Hsieh, and Chen-Hsuan Kuo

Department of Computer Science and Information Engineering,
National Taipei University of Technology, Taipei 106, Taiwan

(Received August 21, 2025; accepted July 10, 2026)

Keywords: dynamic analysis, embedded device, emulation, firmware

Firmware emulation is a key technique for enabling large-scale security analysis of IoT devices and sensing systems without requiring physical hardware. However, existing emulation frameworks often suffer from instability and low success rates owing to mismatches between real device execution environments and virtualized systems, particularly across heterogeneous firmware with different initialization and runtime dependencies. In this study, we investigate failure behaviors in FirmAE, a widely used IoT firmware emulation framework, and identify recurring issues in Boot, Kernel, Network, and nonvolatile random access memory (NVRAM) execution stages. On the basis of this analysis, we propose a set of rule-based improvement strategies that refine boot configurations, Quick Emulator (QEMU) execution parameters, network settings, and NVRAM handling mechanisms to improve emulation stability across diverse firmware types. The main limitation of existing approaches is their reliance on heuristic and incomplete device-specific handling, which reduces robustness when applied to unseen firmware. The proposed method addresses this issue by systematizing failure patterns into actionable repair rules, enabling more consistent emulation behavior. Although the proposed method improves emulation robustness, one limitation of this study is that part of the proposed repair process still depends on manual inspection of firmware logs and runtime behavior, which may limit scalability in fully automated large-scale deployment scenarios. Experimental results on real-world firmware images collected from multiple vendors and architectures demonstrate that the proposed approach improves emulation robustness and enhances the ability to successfully execute previously failing firmware instances, supporting more reliable IoT firmware security analysis.

1. Introduction

With the rapid deployment of IoT technologies in smart homes, industrial automation, healthcare systems, and surveillance applications, large-scale interconnected devices have become a fundamental component of modern sensing and control infrastructures. Many IoT systems rely on networked sensing devices such as cameras and embedded monitoring nodes,

*Corresponding author: e-mail: jykuo@ntut.edu.tw
https://doi.org/10.18494/SAM5910

ISSN 0914-4935

which continuously collect and transmit environmental data for further processing and decision-making. However, the increasing connectivity and deployment scale of such devices have also introduced significant security concerns. Common vulnerabilities include unauthorized access, buffer overflows, backdoors, and weak authentication mechanisms.[1–3] Owing to the resource-constrained nature of IoT devices and the infrequent update cycles from vendors, many vulnerabilities remain unpatched for extended periods, posing serious risks to data integrity, privacy, and system reliability.

From a system perspective, IoT sensing environments are typically composed of heterogeneous devices, including sensing nodes (e.g., network cameras) and edge communication devices (e.g., routers and gateways), which collectively form a distributed sensing and data transmission architecture. The reliability of such systems is heavily dependent on firmware, which governs sensing operations, communication protocols, and hardware control logic. As a result, firmware security plays a critical role in ensuring the correctness and robustness of IoT sensing applications.

To address firmware-level security challenges, firmware analysis has become an essential approach for vulnerability discovery. Existing methods can be broadly categorized into static and dynamic analysis. Static analysis inspects firmware structure and code without execution, enabling the detection of issues such as hardcoded credentials and hidden backdoors. However, it may fail to capture runtime behaviors. In contrast, dynamic analysis executes firmware in emulated or physical environments to observe runtime behavior, allowing detection of vulnerabilities such as memory corruption and command injection.

Fuzz testing is a widely adopted dynamic analysis technique that generates random or mutated inputs to trigger abnormal system behaviors such as crashes or hangs.[4] Compared with manual testing, fuzzing significantly improves test coverage and automation efficiency, making it particularly suitable for IoT firmware with insufficient input validation. In recent studies, the integration of large language models (LLMs) has been further explored to improve fuzzy input generation and target exploration strategies.[5–8]

Owing to the heterogeneity and large scale of IoT devices, physical testing environments are often impractical. Firmware emulation has therefore emerged as a key enabling technology for large-scale IoT firmware analysis. Firmware emulation enables execution and testing without requiring physical hardware by emulating the kernel, file system, and hardware interfaces of target devices. This approach significantly reduces cost while improving scalability and experimental coverage. Environments based on emulation are widely used in automated fuzzing pipelines for large-scale security testing of IoT systems.

However, achieving stable and accurate firmware emulation remains challenging because of inconsistencies between real-world device environments and virtualized execution platforms. In this study, we focus on improving the firmware emulation process within FirmAE,[9] a widely used emulation framework derived from Firmadyne.[10] Although FirmAE significantly improves emulation success rates, it still suffers from failures caused by differences in firmware architecture, initialization procedures, and system dependencies across vendors. These failures may lead to kernel panic, incomplete boot processes, or network service malfunctions. Furthermore, certain firmware heavily

relies on nonvolatile random-access memory (NVRAM),[11,12] and device-specific runtime configurations that are not fully reproduced in generic emulation environments.

To address these limitations, we conduct a systematic analysis of FirmAE emulation failures and categorize them into different failure domains. On the basis of observed error patterns during boot and runtime executions, we propose a set of rule-based improvements targeting boot configuration, QEMU,[13] execution parameters, network environment setup, and NVRAM handling mechanisms.[11,12] These rules are intended to improve compatibility across heterogeneous firmware images and enhance overall emulation stability.

The proposed improvement strategy is implemented by modifying key components of the FirmAE pipeline, including the Arbitration module, the run.sh script, and the firmware image configuration. The original FirmAE framework introduces an Arbitration mechanism to handle emulation failures across four domains: Boot, Kernel, Network, and NVRAM. As shown in Fig. 1, the firmware is first extracted and processed to generate a firmware image suitable for emulation, followed by execution in a QEMU-based virtualized environment. During execution, the Arbitration module applies corrective actions when failures are detected. Although this mechanism improves adaptability, its effectiveness is limited by manually defined heuristics and incomplete device-specific configurations. By refining these Arbitration rules, we further enhance the robustness and success rate of firmware emulation.

Firmadyne,[10] an earlier framework for automated firmware emulation, provides large-scale analysis capabilities but suffers from a relatively low success rate (16.28%)

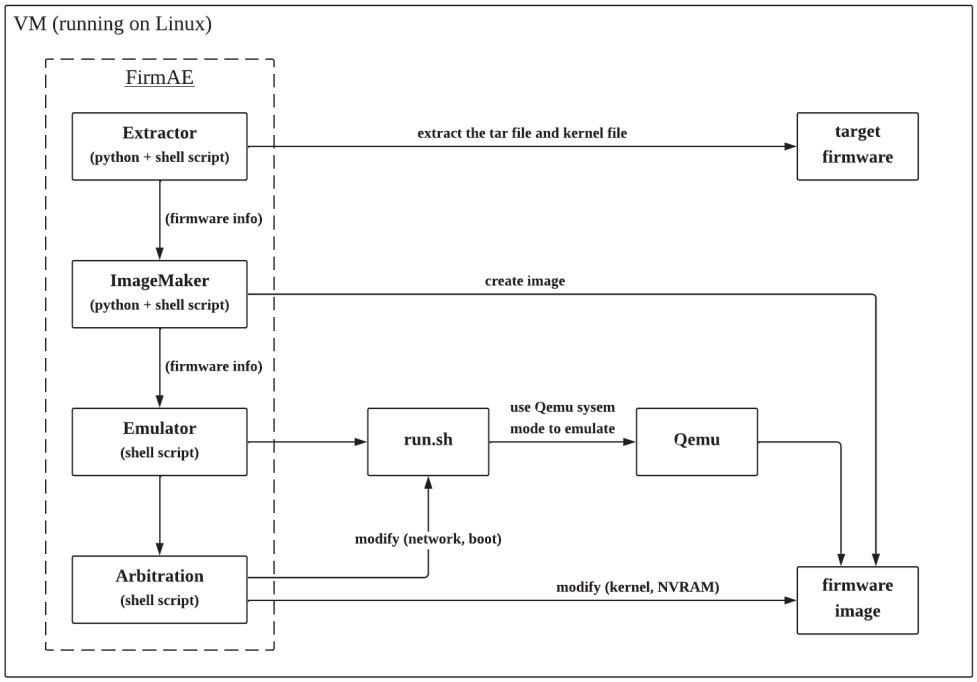

Fig. 1. FirmAE architecture.

owing to limited handling of device-specific execution conditions. FirmAE improves upon this limitation by introducing heuristic-based Arbitration strategies, achieving a significantly higher success rate (79.36%) and enabling successful execution of 892 firmware images, along with the detection of 320 known vulnerabilities and 23 newly identified vulnerabilities, including 12 zero-day cases. However, existing approaches still lack systematic rule generalization and adaptability across diverse firmware families.

In recent years, several firmware emulation frameworks have been proposed for different application scenarios. Firmadyne[10] and FirmAE[9] primarily target Linux-based embedded IoT firmware and support large-scale dynamic analysis through full-system emulation. In contrast, FirmWire is designed for smartphone baseband firmware and focuses on cellular protocol analysis, whereas Avatar² adopts a hardware-assisted hybrid execution model that combines physical devices with emulation to facilitate embedded system analysis. Owing to their different target platforms, execution models, and design objectives, these frameworks address different research problems. Therefore, rather than proposing a new firmware emulation framework, we focus on improving the robustness and adaptability of FirmAE by systematically analyzing representative emulation failures and developing practical repair rules for heterogeneous IoT firmware.

2. Data, Materials, and Methods

In this study, we focus on FirmAE and introduce modifications to its modules to address encountered emulation failures.

2.1 Proposed architecture

The proposed system architecture of this study is illustrated in Fig. 2. The original FirmAE system implements four types of Arbitration, each handling different types of emulation issue. After analyzing FirmAE's Arbitration mechanisms, we identify specific emulation failure problems and propose modifications to the Arbitration module, run.sh script, and firmware images to resolve these failures.

During the system boot phase, we found that some devices encounter kernel panic due to the rdinit parameter in the run.sh startup script. The rdinit parameter is not universally applicable to all devices and must be determined on the basis of hardware support. To resolve this issue, we modified the boot parameters in run.sh and manually specified the appropriate boot files. Additionally, we observed that some devices reported missing kernel modules in their boot logs. This issue arises because the original FirmAE system only includes a limited set of generic modules rather than device-specific modules. To address this, we added the missing modules to the image file, improving compatibility. During network configuration, FirmAE assigns a default 192.168.0.1 Internet Protocol (IP) address to all emulated firmware instances to avoid network-related failures. However, not all devices use this network configuration. We analyzed the internal firmware configuration files and startup scripts to dynamically adjust the network

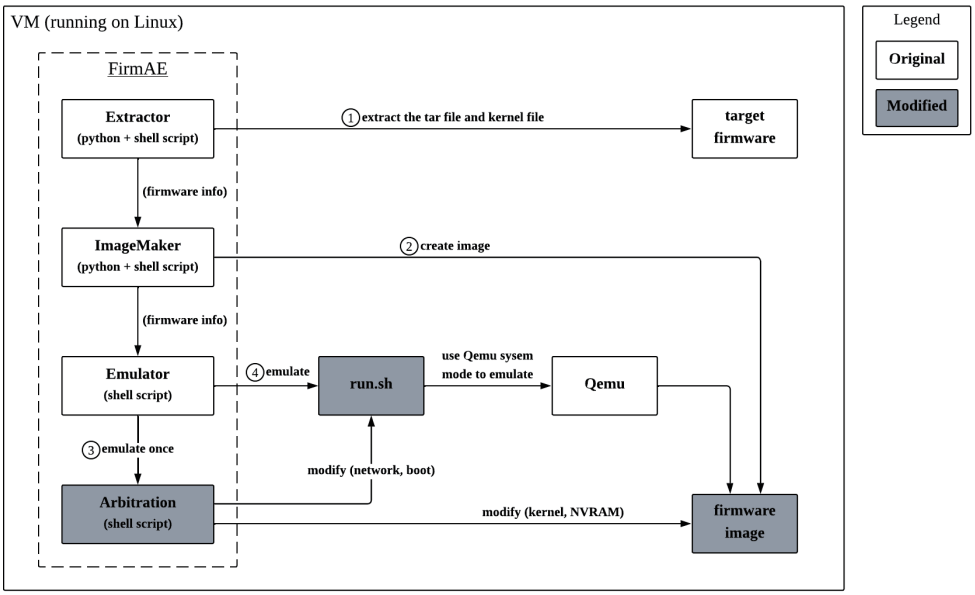

Fig. 2. System architecture.

settings in run.sh on the basis of actual requirements, modifying the host-side IP configuration and the firmware-side IP address in the image file.

We also found that NVRAM Arbitration sometimes interferes with native operations. To address this issue, manual module supplementation and disabling NVRAM Arbitration were adopted to ensure the correct execution of firmware emulation. Additionally, we improved FirmAE's firmware emulation capability by addressing the four types of Arbitration-related issues. To enhance the firmware emulation process, we modified the Arbitration module, run.sh script, and firmware images.

2.1.1 Boot arbitration

The first Arbitration in FirmAE is designed to correct the execution order of "init files" during the firmware boot process. In this process, FirmAE identifies and adjusts the execution sequence of "init files" for different firmware, ensuring that the emulation environment aligns with actual device behavior to prevent emulation failures caused by incorrect boot sequences. In addition to searching for init-related files, the system also processes common system directories, such as proc, tmp, and var. By correcting these commonly used system directories, the firmware can properly access these locations during emulation or execution, reducing the likelihood of system failures.

In this study, we identified two primary issues that prevent the system from booting properly. The first issue arises because the default startup script used by FirmAE is not sufficient for all firmware scenarios. Many manufacturers customize the naming and structure of firmware files within their products, and these modifications often do not conform to general startup logic. As

a result, FirmAE fails to locate or execute the corresponding files correctly during execution, ultimately leading to kernel panic.

The second issue is related to parameter settings in the startup command. FirmAE uses the rdinit parameter by default to specify the startup script. However, not all firmware supports the rdinit parameter. Some embedded system firmware only accepts the init parameter for booting. In cases where the rdinit parameter cannot be used, the system enters an unbootable state, which in turn leads to kernel panic.

To address these issues, we propose improvements to enhance the boot success rate during system emulation. Instead of using FirmAE's default startup script, the startup parameters are manually adjusted on the basis of the file structure and naming convention of each firmware. A more universal "init" parameter is selected to replace "rdinit", and the firmware files are analyzed to determine the appropriate boot file or program that should be used. This approach aims to prevent kernel panic, improving the stability and success rate of the boot process in firmware emulation.

2.1.2 Network arbitration

The second Arbitration in FirmAE focuses on resolving network configuration anomalies. During the emulation process, FirmAE automatically detects and fixes network configuration errors that may affect emulation, ensuring that the emulated device can properly connect to the network.

IP aliasing,[14,15] is a technique that allows multiple IP addresses to be assigned to the same network interface. This configuration enables a device to set multiple IP addresses on a single network card, which can belong to the same subnet or different subnets. FirmAE identified that in Firmadyne's implementation, each detected IP address was assigned to a separate interface, and routing rules were configured for each. To address this, FirmAE modifies the process by configuring only one interface and one IP address, using the default routing settings without additional configurations, as shown in Fig. 3, thereby improving the IP aliasing issue.

In some firmware, the network IP address is dynamically assigned by a Dynamic Host Configuration Protocol (DHCP) server. However, the emulation environment may lack such devices, preventing the firmware from obtaining an IP address properly. To resolve this issue, the method shown in Fig. 4 is used to forcefully configure a network interface and manually assign an IP address, ensuring that the firmware can operate correctly within the emulated environment and maintain network communication, thereby preventing network failures caused by the absence of a DHCP server.

The process of "Configuring Network Interface" begins by using the "brctl" command to create a virtual bridge interface, br0, which is primarily used to forward traffic between multiple physical network interfaces. The br0 interface is then assigned the IP address 192.168.0.1 and activated, typically serving as the main IP address or gateway address for the bridged network. The physical network interface, eth0, is then added to br0, allowing br0 to manage and control all network traffic passing through eth0. Finally, the IP address of eth0 is set to 0.0.0.0 and activated, meaning that eth0 no longer directly handles network communication but instead

```
01  TAPDEV_0=tap${IID}_0
02  HOSTNETDEV_0=${TAPDEV_0}
03  echo "Creating TAP device ${TAPDEV_0}..."
04  sudo tunctl -t ${TAPDEV_0} -u ${USER}
05
06  echo "Bringing up TAP device..."
07  sudo ip link set ${HOSTNETDEV_0} up
08  sudo ip addr add 192.168.0.2/24 dev ${HOSTNETDEV_0}
```

Fig. 3. Program for handling IP alias.

```
01  if [ ${ACTION} == "default" ]; then
02     ${BUSYBOX} brctl addbr br0
03     ${BUSYBOX} ifconfig br0 192.168.0.1
04     ${BUSYBOX} brctl addif br0 eth0
05     ${BUSYBOX} ifconfig eth0 0.0.0.0 up
```

Fig. 4. Program for configuring network interface.

delegates all traffic management to br0. This configuration approach is commonly applied in container networking, virtualization environments, or scenarios requiring multi-network interface coordination, enabling centralized network traffic management and forwarding.

In some firmware, the Virtual Local Area Network (VLAN) functionality is commonly used to isolate different subnets and prevent network interference between different groups. However, FirmAE discovered that Firmadyne does not handle VLAN configurations properly in its emulation environment, leading to failures during the emulation process. To address this issue, FirmAE adopts the method of improving VLAN handling by correctly configuring VLANs to emulate a real network environment.

Some firmware, for network security reasons, explicitly configures firewall rules, which often block web access to the firmware in an emulated environment, preventing proper operation. To address this issue, FirmAE adopts a method where a script continuously checks for the presence of firewall rules during the emulation process. If firewall rules are detected, they are removed, ensuring that the firmware's web interface can be accessed properly.

In this study, we found that FirmAE assigns the default IP address 192.168.0.1 to the network interface (e.g., eth0) of the firmware during configuration. However, this default setting does not apply to all firmware. Some manufacturers design custom network configurations for their products, and these firmware images expect to use specific IP addresses upon startup rather than the 192.168.0.1 assigned by FirmAE. When the firmware cannot match the preset IP address, network functionality fails, leading to emulation failures. In this study, we also discovered that some firmware automatically creates network interfaces and assigns IP addresses during the boot process, without relying on external network configuration tools or scripts. In such cases, FirmAE's forced default IP configuration may interfere with the firmware's normal operation, preventing the emulation process from running correctly.

To address these issues, we developed an improvement method aimed at enhancing FirmAE's network emulation success rate. First, we conducted a detailed static analysis of the firmware to examine configuration files and startup scripts, determining the firmware's default network settings and expected IP addresses. This includes analyzing network-related configuration files within the firmware, such as /etc/network/interfaces or custom network scripts, to identify the necessary IP addresses and network parameters required during firmware execution. On the basis of the analysis results, the network configuration is dynamically adjusted in the emulation environment to avoid using FirmAE's default IP address 192.168.0.1. Instead, an appropriate IP address is assigned in accordance with the firmware's actual requirements. If the firmware is found to independently create and manage network interfaces, no external network configuration is enforced, ensuring that the firmware operates correctly within the emulated environment.

2.1.3 Kernel arbitration

The third Arbitration in FirmAE addresses compatibility issues between different Kernel versions. Since firmware is typically developed on the basis of various versions of the Linux Kernel, newer firmware often runs on higher Kernel versions, leading to incompatibilities during the emulation process owing to version differences. To solve this problem, FirmAE adopts the most direct approach by providing a newer compiled Kernel, ensuring that the firmware can execute correctly within the emulated environment.

During this process, we found that although the generic Kernel provided by FirmAE meets the basic requirements of most firmware, some firmware cannot function properly owing to dependencies on specific Kernel modules. These modules are not included in the generic Kernel, preventing the firmware from loading necessary functionalities and leading to emulation failures. For example, certain network firmware may require specific network protocol modules, or storage systems may need specialized file system modules. To resolve this issue, we analyze the target firmware to identify the required modules for execution. These modules are then manually provided to the firmware, ensuring compatibility and allowing the firmware to correctly load the necessary modules at startup. This approach is aimed at eliminating emulation failures caused by missing modules.

2.1.4 NVRAM arbitration

The fourth Arbitration in FirmAE is designed to handle the configuration of NVRAM required by firmware. Properly configuring NVRAM values during the emulation process ensures that the returned values do not cause system crashes that could lead to emulation failure. During this process, we found that while FirmAE provides basic NVRAM operations to support the required firmware system operations, some firmware requires additional operations that are overwritten by FirmAE's NVRAM Arbitration, resulting in emulation failure. For such firmware, we propose disabling NVRAM Arbitration, allowing the firmware to execute its native NVRAM operations without interference.

2.2 Firmware emulation failure recovery process

In this study, we improve firmware emulation by following the failure recovery process shown in Fig. 5, addressing issues encountered during the emulation process.

2.2.1 Redirect output

In the first Redirect Output phase, the original log output is also redirected to the Command Line (CMD). This approach allows real-time monitoring of the firmware execution during emulation, making it easier for us to observe the firmware's behavior. Additionally, it enables users to directly input commands into the command line for interaction with the emulated firmware.

2.2.2 Repair rule induction

In the second phase, we analyze the issues encountered during the implementation of Arbitration in FirmAE and conduct an in-depth examination of these problems. During the repair process, we evaluate various factors that may affect the emulation success rate and develop corresponding fixes on the basis of different scenarios. These improvements will then be applied to the firmware emulation environment. Ultimately, we systematically organize these fixes and summarize them into a set of repair rules to address common issues in the emulation process, aiming to enhance the overall success rate and stability of firmware emulation. Additionally, in the second phase, the repaired firmware will be tested to verify whether the emulation is successful. The criteria for determining success include the following.

A. The host side must be able to perform a ping test on the firmware side and receive a successful response.
B. The host side must be able to access the firmware's web service through a browser.

2.2.3 Validate rules

In the third phase, we further verified the applicability and effectiveness of the repair rules summarized in the second phase. Additional firmware was selected and tested using FirmAE for emulation. The repair rules established in the second phase were applied to these firmware images, and the emulation success rate and execution status were compared before and after applying the fixes to evaluate whether these repair rules can be widely applied to different

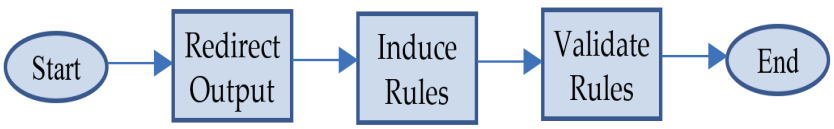

Fig. 5. (Color online) Repair process.

firmware architectures and environments. Successful improvement in the emulation success rate demonstrated the effectiveness and applicability of the proposed repair rules.

3. Implementation and Results

In this section, we present the experimental setup and validation results.

3.1 Implementation environment

The experiments were conducted on the Ubuntu platform, and the experimental hardware specifications are shown in Table 1.

We collected six firmware images from publicly available sources and the FirmAE dataset, originating from different manufacturers and CPU architectures, as shown in Table 2. These firmware images could not be successfully emulated using FirmAE. In Sect. 3.2, we analyze these firmware images, apply necessary fixes, and summarize the techniques used during the emulation process into a set of repair rules.

We verified the repair rules summarized from the experiments in Table 2 by testing publicly available firmware provided by Delta.[16] The goal was to determine whether these rules are also applicable to Delta firmware listed in Table 3. Although the DX3001H9 model appears in both Tables 2 and 3, the firmware versions originate from different years. Therefore, we specifically examine whether the repair rules remain effective across different years of DX3001H9 firmware. Additionally, the DX3021L9 model serves as a completely new verification target that has never been used in previous experiments. This allows further validation of the repair rules' applicability to new models. Through these validations, we aim to confirm the generalizability and effectiveness of the repair rules across different firmware versions and new device models.

3.2 Implementation and analysis of the improvement methods

In this study, we analyzed some firmware emulation issues to develop the proposed repair rules.

3.2.1 Improvement analysis of firmware for model CD8371

This firmware supports a Delta Network Camera device. When attempting to emulate this firmware using FirmAE, it was observed that after a period of emulation, requests from the host

Table 1
Experimental environment specifications.

CPU	Intel(R) Core(TM) i5-12500
RAM	16 GB
SSD	100 GB
OS	Ubuntu 18.04 LTS

Table 2
Experimental firmware repair data.

	Firmware name	Architecture	Type
Experiment 1	CD8371	armel	Network camera
Experiment 2	DX3001H9	armel	Router
Experiment 3	DIR513A	mipsel	Router
Experiment 4	TEW692GR	mipsel	Router
Experiment 5	RAX120	armel	Router
Experiment 6	NBG418N	mipsel	Router

Table 3
Experimental firmware verification data.

	Firmware name	Architecture	Type
Experiment 1	DX3001H9	armel	Router
Experiment 2	DX3021L9	armel	Router

to the emulated firmware did not receive any response. Additionally, the output messages from the preInit.sh initialization script were not found in the logs. Upon investigation, it was found that in the QEMU startup command, the original run.sh script used the rdinit parameter, preventing the preInit.sh script from executing correctly. When the rdinit parameter was replaced with the init parameter, the preInit.sh script and the network configuration script were successfully executed. The corresponding successful configuration messages were also found in the logs. This modification effectively resolved the issue of the invalid boot script configuration, and in subsequent emulation experiments, we applied this approach when encountering similar issues.

3.2.2 Improvement analysis of firmware for model DX-3001H9

This firmware supports a Delta Router device. When attempting to emulate this firmware using FirmAE, it was observed that after a period of emulation, using the preInit.sh script as the init process to boot the system resulted in a kernel panic. This indicates that the boot file or execution order may be incorrect. To address this issue, we first modified the emulation startup command. Initially, the system was booted using rdinit=preInit.sh. After changing the parameter to init=preInit.sh, the system successfully booted. Further analysis of the /etc/inittab file revealed that after booting, the system should execute /etc/init.d/rcS. Therefore, we manually executed /etc/init.d/rcS, allowing the system to complete the initialization process. After executing this script, the system began performing related initialization settings without encountering kernel panic. Additionally, /etc/init.d/rcS handled various startup configurations, including mounting directories and loading necessary system modules for execution.

Further analysis of the system boot logs revealed that during module loading, the logs showed missing network module support for x_tables.ko, xt_tcpudp.ko, and nfnetlink.ko, resulting in missing module warning messages. To resolve this issue, these modules were manually compiled to meet system requirements. After providing the necessary modules, the missing module messages no longer appeared.

Next, multiple network configuration-related messages were observed in the logs, as shown in Fig. 6. Further investigation confirmed that these messages were caused by the system executing the rcS.lan script, which attempted to access a vendor-compiled config executable located in the /usr/sbin/ directory to retrieve network configuration information. However, the returned data was empty.

Subsequently, in the lower section of the script, the system attempted to use this retrieved network information with the ifconfig command, leading to the error message "ifconfig: down: error fetching interface information: Device not found". Since the config executable was custom-compiled by the device manufacturer, we could not directly modify or replace it with other tools to resolve the issue. As a result, the emulation process only allowed error observation without further improvements or repairs to this part of the network configuration process. This finding highlights the challenges in analyzing customized firmware, as vendor-specific tools and configurations can complicate research efforts.

We attempted to modify the rcS.lan script file by setting commonly used values found in standard firmware. Specifically, the LAN variable was changed to eth0, the BR0 variable to br0, the lan_ip variable to 192.168.1.1, and the lan_mask variable to 255.255.255.0. These modifications prevented the script from accessing /usr/sbin/config. After making these changes, the ifconfig-related errors caused by the inability to retrieve network interface data no longer appeared in the logs. After modifying the rcS.lan file and rebooting the system, the network interface still failed to configure successfully, and network connectivity remained unsuccessful. This suggests that modifying the script and setting default values did not improve the emulation of this firmware.

Further observation revealed that the rcS.lan file frequently contained get and set operations, which closely resembled NVRAM access operations. Since FirmAE intercepts these NVRAM operations but does not support this specific firmware's implementation, the firmware could not retrieve the correct configuration values. To address this issue, we attempted to remove the NVRAM-related files used by FirmAE and disable NVRAM Arbitration. After rebooting, the system attempted to read FirmAE's NVRAM files, but upon failing to locate them, it defaulted

```
311  ifconfig: down: error fetching interface information: Device not found
312  [   15.853053] cfg80211: Calling CRDA to update world regulatory domain
313  ifconfig: bad address 'ether'
314  ifconfig: up: error fetching interface information: Device not found
315  ifconfig: up: error fetching interface information: Device not found
316  [   19.000925] cfg80211: Calling CRDA to update world regulatory domain
317  [   22.151939] cfg80211: Calling CRDA to update world regulatory domain
318  ifconfig: bad address 'ether'
319  expr: syntax error
320  expr: syntax error
321  ifconfig: SIOCSIFMTU: No such device
322  ifconfig: SIOCSIFADDR: No such device
323  ifconfig: SIOCSIFADDR: No such device
324  ifconfig: SIOCSIFADDR: No such device
```

Fig. 6. (Color online) Network misconfiguration messages.

to using the system's native operations. After a period of configuration, using the ifconfig command to check the network settings confirmed that the br0 network interface was successfully created, and the IP address was set to 192.168.1.1.

This observation indicated that the network interface did not require intervention from FirmAE's network scripts for this firmware instance. Therefore, the FirmAE network scripts were disabled to avoid network interface conflicts. However, even after these modifications, the host was still unable to ping the firmware. Further investigation revealed that this issue was caused by a Media Access Control (MAC) address mismatch between the br0 and eth0 network interfaces. A MAC address inconsistency can prevent network packets from being correctly routed or responded to, leading to communication failures, Address Resolution Protocol conflicts,[17–19] and bridge interface forwarding failures within the emulated environment.

3.2.3 Improvement analysis of firmware for model DIR513A

This firmware supports a D-Link Router device, which is also one of the failed emulation cases in the original FirmAE dataset. According to recorded data, although the firmware's IP address could be accessed by the host, the web service failed to start successfully. In this experiment, we attempted to emulate the firmware using FirmAE and observe its behavior. During the emulation process, it was found that after a certain period, the host was unable to access the firmware using the default 192.168.0.1 IP address. Additionally, the web service did not respond properly to host requests, preventing further web access operations.

We modified the startup command by setting init=preInit.sh. After successfully booting, the script /etc_ro/rcS was executed. The system logs then displayed a vendor welcome message and network configuration details. The observed IP configuration differed from FirmAE's default IP settings, suggesting that the vendor may have a preferred IP configuration. This discrepancy likely caused the default 192.168.0.1 IP to be inaccessible for this firmware.

To resolve this, the network configuration script within the firmware was modified to match the firmware's internal settings, as shown in Fig. 7. Additionally, the host-side IP configuration script was updated, ensuring that the host and firmware are within the same subnet, allowing the host to access the firmware. After modifying these configurations, the firmware was re-emulated, and the host successfully accessed the assigned IP and retrieved the web service provided by the firmware through a browser. The web UI interface and operation process align with the content provided in the manufacturer's operation manual. Through this experiment, we demonstrated that the proposed correction method effectively repairs network functionality within the emulated firmware.

```
01  if [ ${ACTION} == "default" ]; then
02      ${BUSYBOX} brctl addbr br0
03      ${BUSYBOX} ifconfig br0 192.168.0.50
04      ${BUSYBOX} brctl addif br0 eth0
05      ${BUSYBOX} ifconfig eth0 0.0.0.0 up
```

Fig. 7. Modified network interface configuration script.

3.2.4 Improvement analysis of firmware for model TEW-692GR

This firmware supports a Trendnet router device, which is also one of the failed emulation cases in the original FirmAE dataset. According to recorded data, although the firmware's IP can be accessed from the host side, its web service fails to start successfully. In this experiment, we used FirmAE to emulate this firmware and observe its behavior. During the emulation process, it was found that, similar to the DIR513A model, the host could not ping the firmware, nor could it access the web service. Next, we modified the boot command by setting init=preInit.sh. After successfully booting, the system first executed a script named /etc_ro/rcS in the filesystem. Upon execution, a message appeared in the firmware logs. Further analysis revealed that this message was generated by an executable file named goahead. The message indicated that before executing goahead, an executable file named nvram_daemon should be executed first. The cause of this issue was traced back to the original implementation of the preInit.sh script in FirmAE, which attempts to start certain services required by the firmware.

To address this issue, we searched the filesystem for files related to nvram_daemon. Eventually, the execution command for nvram_daemon was found in the /etc_ro/rcS file. After locating this script, the service startup command in preInit.sh was commented out, and the run_service.sh execution command was added immediately after the nvram_daemon execution line in the /etc_ro/rcS script to start the goahead service. After modifying the script and re-emulating the firmware, the error message no longer appeared. Instead, network configuration messages were displayed, including setting the IP to 192.168.0.50. A notable aspect of this experiment is that the firmware's network interface does not require external intervention. The firmware automatically configures the network interface and assigns the IP address 192.168.0.50 for external access. In this case, the host side only needs to remain on the same subnet as the firmware without additional network configuration, preventing potential conflicts.

3.2.5 Improvement analysis of firmware for model RAX120

This firmware supports a Delta WiFi AP device. During the emulation of this firmware using FirmAE, it was observed that after a period of emulation, an error message appeared. This message indicates that a required file is missing, preventing execution. To investigate why the system attempts to execute this file, we analyzed the firmware files, and it was discovered that the /etc/inittab file contains an instruction to execute the /etc/init.d/rcS script after booting for system initialization. However, in the extracted filesystem, this file was missing, preventing further execution and leading to emulation failure. This case also demonstrates that firmware does not necessarily include all files required for complete system execution, which poses challenges in firmware emulation.

3.2.6 Improvement analysis of firmware for model NBG-418N

This firmware supports a Zyxel router device, which is also one of the failed emulation cases in the original FirmAE dataset. According to recorded data, although the firmware's IP can be accessed from the host side, its web service fails to start successfully. To analyze this failure

case, we used FirmAE to emulate this firmware and observed its execution behavior. During the emulation process, it was found that, similar to the DIR513A model, the host could not ping the firmware, nor could it access the web service. Next, we attempted to boot the system using preInit.sh. After booting, it was observed that the contents of the /etc directory in this firmware differed from those commonly found in other firmware. It lacked common initialization scripts such as init, preinit, and rcS. Additionally, the /etc/inittab file did not define any initialization scripts to be executed after booting. However, a file named linuxrc was found in the root directory, which also serves as an initialization script. Therefore, we executed this file for initialization and then manually executed /usr/sbin/httpd to start the web service in the firmware. After execution, the system began setting up relevant configurations.

Next, we further analyzed the IP-related configurations. Therefore, the firmware's IP address was set to 192.168.0.50, and the host's IP was configured within the same subnet. As a result, the host was able to successfully ping the firmware and access the firmware's web service via a browser. Through this analysis, we found that this firmware has a unique characteristic: as long as its IP is configured within the 192.168.0 subnet, it can be successfully accessed by the host, and the web service can also be accessed normally. This behavior may be related to the network settings defined by the manufacturer.

3.3 Verification and analysis of the improvement methods

3.3.1 Implementation results

The experimental results of the firmware mentioned in Sect. 3.2 are shown in Table 4. According to the experimental results, we successfully repaired four out of six failed firmware emulations, achieving a repair rate of 66.7% (4/6) (Table 5), significantly improving the emulation success rate. Throughout the experiment, we conducted in-depth analysis and adjustments based on the causes of emulation failures; the proposed repair rules were effectively applied to firmware emulation environment configuration and troubleshooting.

After applying the repair rules, the firmware successfully executed system functions and allowed web-based access to the device's services, thereby validating the effectiveness of the proposed repair rules. Furthermore, these repair rules not only improved the firmware emulation success rate during the experiment but also provided a practical foundation for further firmware emulation research, highlighting the application value of this study in the field of firmware emulation.

Table 4
Statistics of repair rule application frequency.

Arbitration	Rule number	Number of applications
Boot	Rule 1	6
	Rule 2	1
Kernel	Rule 3	3
NVRAM	Rule 4	3
Network	Rule 5	5
	Rule 6	3
	Rule 7	3

Table 5
Repair results.

Firmware name	Type	Architecture	FirmAE result	Our method
CD8371	Network camera	armel	Fail	Fail
DX3001H9	Router	armel	Fail	Success
DIR513A	Router	mipsel	Fail	Success
TEW692GR	Router	mipsel	Fail	Success
RAX120	Router	armel	Fail	Fail
NBG418N	Router	mipsel	Fail	Success

- Rule 1: Kernel Panic Boot Recovery Rule

Condition: The system encounters a kernel panic during the boot phase, and the issue is related to the initialization script configuration.

Solution: First, check whether the boot command uses the rdinit parameter and replace it with the init parameter. This ensures that the system follows the standard initialization process during boot. Next, set preInit.sh as the startup script. This script handles essential pre-configuration tasks, such as mounting the filesystem, setting kernel parameters, or starting necessary services. Additionally, once the system successfully boots, it should further execute the initialization scripts specified in the /etc/inittab file. These scripts typically define the steps in the system initialization process, such as starting system-related services, configuring network parameters, and loading additional modules, ensuring that the system reaches a stable operational state. This approach effectively resolves kernel panic issues caused by misconfigured or mismatched initialization scripts and improves the system's boot success rate.

- Rule 2: System Module Completion Rule for Missing Components

Condition: During the system initialization process, certain modules are missing, preventing the initialization process from proceeding normally or limiting system functionality.

Solution: First, record the names of the missing modules encountered during the initialization process. Then, on the basis of the system architecture (e.g., ARM or x86) and module dependencies, obtain or compile the missing modules, ensuring compatibility with the target kernel version. After completing the module supplementation, restart the system and check whether the missing module messages continue to appear in the logs. Ensure that the newly added modules are correctly loaded and functioning properly during execution to prevent new issues caused by missing or faulty modules.

- Rule 3: NVRAM Support Rule

Condition: During the initialization process, NVRAM support is found to be insufficient, preventing successful emulation.

Solution: The NVRAM Arbitration implementation in FirmAE may sometimes override the native system and lack sufficient support, preventing proper responses to system calls. In such cases, the recommended approach is to disable NVRAM Arbitration and use the native system's NVRAM settings or supplement the necessary NVRAM values to ensure proper support.

- Rule 4: Script Execution Order Correction Rule

Condition: During the system initialization process, the system log records indicate an error in the script execution order.

Solution: When an error message regarding script execution order appears in the system log, the first step is to examine the system log and record the name of the erroneous script along with the contextual information of the execution failure to pinpoint the issue. Next, analyze the relevant scripts to determine their dependencies and verify whether the execution order meets the system's requirements, especially checking if any critical resources were not properly initialized or were executed before their dependent scripts. If the execution order does not conform to system requirements, adjustments should be made to correct the script execution sequence. After making the necessary changes, restart the system and conduct tests to ensure that the script execution order aligns with system requirements and that the error message no longer appears.

- Rule 5: Network Configuration IP Address Correction Rule

Condition: During the network configuration phase, the system's assigned network IP address is inconsistent with the default IP address (192.168.0.1), leading to network connection issues or configuration errors.

Solution: When detecting an inconsistency between the assigned network IP address and the default address, the following steps should be taken to ensure proper network configuration and system operation. First, check the network IP address that the firmware system is set to use and verify whether it differs from the default IP address (192.168.0.1). This can be done using the ifconfig or ip address show command to inspect the IP address of the network interface. Next, on the basis of the actual requirements, modify the system's network configuration script network.sh to set the system IP address to an appropriate one that is compatible with the operating environment and within the same subnet. In addition to configuring the firmware-side IP address, the host-side network IP must also be adjusted to remain within the same subnet as the firmware. This can be done by manually setting the host's IP address. After making these adjustments, network connectivity should be tested using commands such as ping to verify that the configuration is correct. Additionally, the system log should be checked for any related error messages.

- Rule 6: Network Interface MAC Address Correction Rule

Condition: During the network configuration phase, the MAC addresses of the bridge interface br0 and the physical interface eth0 are inconsistent.

Solution: First, stop the network services related to br0 and delete the br0 interface. Next, recreate the br0 bridge interface and adjust its MAC address to match that of eth0. After completing the configuration, restart the network services or manually enable br0 to apply the changes. Finally, perform tests to ensure proper network communication between the host and the firmware.

- Rule 7: Firmware and FirmAE Network Configuration Conflict Correction Rule

Condition: During the network configuration phase, the firmware's initialization script automatically configures the network interface, and this configuration may conflict with FirmAE's network settings.

Solution: When it is discovered that the firmware's initialization script actively configures the network interface, FirmAE's network settings should be disabled to prevent conflicts that may cause network malfunctions. First, examine the firmware's initialization scripts, such as those under /etc/init.d/ or the /etc/network/interfaces file, to verify whether they contain network configuration logic, including IP address assignment, subnet mask settings, or other network-related configurations. If the firmware has an automatic network configuration mechanism, FirmAE's network configuration should be disabled. This can be achieved by modifying FirmAE's scripts and commenting out the relevant network configuration code. After making these modifications, restart the emulation environment and observe whether the firmware's built-in network configuration executes correctly. Additionally, test network communication by using tools such as ping to verify connectivity between the host and the firmware.

3.3.2 Verification

In this section, we will use the firmware listed in Table 3 to verify the effectiveness and general applicability of Rules 1 to 7, which were derived from the experiments presented in Table 2.

A. Verification of DX3001H9 firmware

This firmware supports a Delta router device. During the emulation of this firmware using FirmAE, we observed that a kernel panic occurred after a period of emulation when the preInit.sh script was used as the initialization program during the init phase. This matches the condition described in Rule 1. Therefore, we applied Rule 1, changing the boot parameter from rdinit to init and using the /etc/init.d/rcS file as the post-boot initialization script.

Next, the system log indicated that certain network-related modules were missing. Following the solution described in Rule 2, the missing modules were recompiled and provided for the firmware. After supplementation, the related error messages no longer appeared. When the initialization script executed the network configuration script, the same issue as described in Sect. 3.2.2 occurred. During the network configuration process, the system attempted to access NVRAM values, but NVRAM support was unavailable. To address this, Rule 3 was applied by

disabling NVRAM Arbitration, allowing the system to use its native configuration values. After implementing this modification and rebooting, the system successfully completed the network interface and IP configuration. Following Rule 7, since the firmware was already handling its own network configuration, we did not need to manually configure the firmware's IP address. Instead, by applying Rules 5 and 6, the host's IP was configured to be within the same subnet as the firmware, and the MAC addresses of br0 and eth0 were synchronized. This approach enabled successful emulation.

B. Verification of DX3021L9 firmware

This firmware supports a Delta router device. During the emulation of this firmware using FirmAE, we observed that a kernel panic occurred after a period of emulation when the preInit.sh script was used as the initialization program during the "init" phase. This matches the condition described in Rule 1. Therefore, we applied Rule 1, and after modifying the settings and rebooting, the system messages appeared correctly, and the kernel panic no longer occurred. After successfully booting, the initialization script "/etc/init.d/rcS" was executed. Similar to previous cases, the system log indicated that network-related modules were not correctly loaded. Therefore, we applied Rule 2, and the missing modules were supplemented to resolve this issue.

Additionally, this firmware exhibited the same problem as that for DX-3001H9, where during the network configuration process, the system attempted to access NVRAM values but received no response, leading to network configuration failure. To address this, we applied Rule 3 and disabled NVRAM Arbitration. After rebooting and executing the initialization script, the ifconfig command was used to check the firmware's network settings, revealing that the firmware had automatically created a network interface and successfully assigned the IP address 192.168.254.254. Given this situation, Rule 7 was applied to disable the network.sh script to prevent interference with the native system's network configuration. Additionally, Rule 5 was applied to configure the host's IP address as 192.168.254.253, ensuring it remained within the same subnet. These modifications enabled the successful emulation of the firmware.

4. Discussion

In Sect. 3.3.2, the results indicate that the emulation repair rules derived from the experiments in Sect. 3.2 were successfully applied to the firmware listed in Table 3. These repair rules not only resolved potential issues during the emulation process but also ensured the normal execution of the firmware within the emulation environment, ultimately achieving successful emulation of the target firmware. This validation demonstrates the practicality and reliability of the repair rules and further indicates that these rules demonstrate good generalizability and applicability across different firmware models and versions.

Table 5 summarizes the repair results of the experimental and validation firmware, whereas Table 6 lists the repair rules applied to each firmware image. According to statistical results, the frequency of application of the seven rules reflects the distribution of various issues encountered during the firmware emulation process and their respective resolution needs. Rule 1 was the most frequently applied rule, with a total of six occurrences, indicating that kernel panic during

Table 6
Applied repair rules for each firmware image.

	Name	Applied rule(s)
Experiment dataset	CD8371	1
	DX3001H9	1, 2, 3, 5, 6, 7
	DIR513A	1, 5
	TEW692GR	1, 4, 5
	RAX120	1
	NBG418N	1
Validation dataset	DX3001H9	1, 2, 3, 5, 6, 7
	DX3021L9	1, 2, 3, 5, 6, 7

the initialization phase is one of the primary challenges in emulation failures. This issue is likely closely related to misconfigured initialization scripts or incorrect system boot parameters. Rule 5 was the second most frequently applied rule, reflecting the high occurrence rate of network-related issues during emulation. Rules 2, 3, 4, 6, and 7, were applied one to three times, suggesting that these issues occur less frequently and may arise only under specific circumstances.

Although the proposed rules substantially improved the emulation success rate, CD8371 and RAX120 could not be successfully repaired because their firmware relied on missing vendor-specific components and incomplete runtime files that could not be reconstructed solely through configuration modifications. In our future work, we will investigate techniques for reconstructing missing vendor-specific runtime components and incomplete firmware files to further improve the emulation success rate. Compared with FirmAE, our method does not redesign the framework, but instead improves its robustness through systematic failure analysis and reusable repair rules. The remaining failures were mainly caused by incomplete firmware packages and unavailable vendor-specific runtime components rather than limitations of the proposed repair rules.

5. Conclusions

In this paper, we proposed a systematic set of improvement rules for firmware emulation in embedded devices to address issues encountered during the emulation process. These rules clearly define applicable solutions for various emulation failure scenarios, including the selection of boot files, the configuration of QEMU startup commands, network environment settings, and the handling of NVRAM. Problems caused by incomplete configurations or mismatched system requirements in firmware emulation can be effectively resolved by applying these rules, providing a more stable execution environment for emulation.

In the experiments, we systematically applied these rules to correct deficiencies in the emulation process, successfully reducing the number of failed emulation cases of four different devices and accurately reproducing their real-world operating conditions. These devices have different hardware architectures and system requirements, demonstrating the broad applicability

and effectiveness of the proposed rules. The proposed approach not only enhances the success rate of emulation but also provides concrete methodological references and technical support for future dynamic analysis and vulnerability detection of embedded devices.

Although the proposed repair rules significantly improve firmware emulation reliability, several limitations remain. Some repair procedures, such as boot parameter selection, kernel module supplementation, and NVRAM configuration adjustments, still require manual analysis on the basis of firmware-specific characteristics. Therefore, the current approach may be less efficient when applied to large-scale firmware collections. In future work, automated log-analysis and rule-induction mechanisms to classify emulation failures and automatically recommend or apply appropriate repair strategies can be added to the framework. Such automation is expected to improve the scalability of firmware emulation and further facilitate large-scale security analysis, vulnerability assessment, and the development and validation of IoT sensing systems without requiring physical devices. The proposed repair rules can also serve as practical guidelines for future automated firmware repair systems.

Author Contributions

The authors confirm contributions to the paper as follows: study conception, funding acquisition, supervision and methodology, Jong-Yih Kuo; writing—original draft preparation, Yu-Quan Wang and Ti-Feng Hsieh; writing—review and editing, Cheng-Hsuan Kuo.

Acknowledgments

This work was supported by the National Science and Technology Council under grant number 113-2221-E-027-126-MY3.

Conflicts of Interest

The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1 F. Meneghello, M. Calore, D. Zucchetto, M. Polese, and A. Zanella: IEEE Internet Things J. **6** (2019) 8182. https://doi.org/10.1109/JIOT.2019.2935189
2 N. Neshenko, E. Bou-Harb, J. Crichigno, G. Kaddoum, and N. Ghani: IEEE Commun. Surv. Tutor. **21** (2019) 2702. https://doi.org/10.1109/COMST.2019.2910750
3 M. Yu, J. Zhuge, M. Cao, Z. Shi, and L. Jiang: Future Internet **12** (2020) 27. https://doi.org/10.3390/fi12020027
4 J. Li, B. Zhao, and C. Zhang: Cybersecurity **1** (2018) 1.
5 W. Wang, K. Liu, A. R. Chen, G. Li, Z. Jin, G. Huang, and L. Ma: Python Symbolic Execution with LLM-powered Code Generation. arXiv preprint arXiv:2409.09271 (2024).
6 C. S. Xia, M. Paltenghi, J. Le Tian, M. Pradel, and L. Zhang: Proc. IEEE/ACM 46th Int. Conf. Software

Engineering **126** (2024) 1. https://doi.org/10.1145/3597503.3639121
7 J. Xu, J. Xu, T. Chen, and X. Ma: Proc. 2024 IEEE 24th Int. Conf. Software Quality, Reliability and Security (QRS) (2024) 228. https://doi.org/10.1109/QRS62785.2024.00031
8 J. Yu, Y. Shao, H. Miao, J. Shi, and X. Xing: arXiv preprint arXiv:2409.14729 (2024). https://doi.org/10.48550/arXiv.2409.14729
9 D. Kim, E. Kim, M. Kim, Y. Jang, and Y. Kim: IEEE Secur. Priv. **19** (2021) 26. https://doi.org/10.1109/MSEC.2021.3076226
10 D. D. Chen, M. Woo, D. Brumley, and M. Egele: NDSS (2016) 1. https://doi.org/10.14722/ndss.2016.23415
11 B. Van Essen, R. Pearce, S. Ames, and M. Gokhale: Proc. 26th IEEE Int. Parallel Distrib. Process. Symp. (2012) 703. https://doi.org/10.1109/IPDPS.2012.69
12 S. Pelley, T. F. Wenisch, B. T. Gold, and B. Bridge: Proc. VLDB Endowment **7** (2013) 121. https://doi.org/10.14778/2732228.2732231
13 F. Bellard: Proc. USENIX Annual Technical Conf. (FREENIX Track) **41** (2005) 10.
14 K. Keys: ACM SIGCOMM Comput. Commun. Rev. **40** (2010) 50. https://doi.org/10.1145/1672308.1672318
15 M. H. Gunes and K. Sarac: Proc. IEEE Global Internet Symp. (2007) 19. https://doi.org/10.1109/GI.2007.4301425
16 Download Center: https://downloadcenter.deltaww.com/en-US/DownloadCenter (accessed February 2025).
17 D. Hercog and D. Hercog: Communication Protocols: Principles, Methods and Specifications (2020) 321. https://doi.org/10.1007/978-3-030-50405-2_19
18 M. G. Gouda and C.-T. Huang: Comput. Netw. **41** (2003) 57. https://doi.org/10.1016/S1389-1286(02)00326-2
19 E. León, B. S. Reyes Daza, and O. J. Salcedo Parra: Proc. Future Data and Security Engineering Conf. (2015) 72. https://doi.org/10.1007/978-3-319-26135-5_6