Battling against Protocol Fuzzing: Protecting Networked Embedded Devices from Dynamic Fuzzers

PUZHUO LIU, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, China
YAOWEN ZHENG*, Nanyang Technological University, Singapore
CHENGNIAN SUN, Cheriton School of Computer Science, University of Waterloo, Canada
HONG LI, Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences, China
ZHI LI, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, China
LIMIN SUN, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, China

Networked Embedded Devices (NEDs) are increasingly targeted by cyberattacks, mainly due to their widespread use in our daily lives. Vulnerabilities in NEDs are the root causes of these cyberattacks. Although deployed NEDs go through thorough code audits, there can still be considerable exploitable vulnerabilities. Existing mitigation measures like code encryption and obfuscation adopted by vendors can resist static analysis on deployed NEDs, but are ineffective against protocol fuzzing. Attackers can easily apply protocol fuzzing to discover vulnerabilities and compromise deployed NEDs. Unfortunately, prior anti-fuzzing techniques are impractical as they significantly slow down NEDs, hampering NED availability.

To address this issue, we propose Armor—the first anti-fuzzing technique specifically designed for NEDs. First, we design three adversarial primitives—delay, fake coverage, and forged exception—to break the fundamental mechanisms on which fuzzing relies to effectively find vulnerabilities. Second, based on our observation that inputs from normal users consistent with the protocol specification and certain program paths are rarely executed with normal inputs, we design static and dynamic strategies to decide whether to activate the adversarial primitives. Extensive evaluations show that Armor incurs negligible time overhead and effectively reduces the code coverage (e.g., line coverage by 22%-61%) for fuzzing, significantly outperforming the state-of-the-art.

CCS Concepts: • **Security and privacy** → **Software security engineering**; Artificial immune systems; • **Computer systems organization** → **Embedded software**; • **Networks** → Security protocols.

*Corresponding Author

Authors' addresses: Puzhuo Liu, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, liupuzhuo@iie.ac.cn; Yaowen Zheng, Nanyang Technological University, Singapore, yaowen.zheng@ntu.edu.sg; Chengnian Sun, Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, cnsun@uwaterloo.ca; Hong Li, Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, lihong@iie.ac.cn; Zhi Li, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, lizhi@iie.ac.cn; Limin Sun, Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, CAS; School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China, sunlimin@iie.ac.cn.

ACM 1049-331X/2024/1-ART
https://doi.org/10.1145/3641847

Additional Key Words and Phrases: Internet of Things, protocol fuzzing, anti-fuzzing

1 INTRODUCTION

The widespread utilization of networked embedded devices (NEDs), such as WiFi routers, programmable logic controllers (PLCs), and robots, has undeniably revolutionized the efficiency and convenience of our daily lives. These NEDs have played an important role in facilitating the emergence of intelligent applications such as the Internet of Things (IoT), the Industrial Internet, and the Intelligent Connected Vehicle. However, this progress has also brought increased susceptibility to NEDs from cyberattacks [56]. These cyberattacks targeting NEDs can result in severe consequences, ranging from stealing sensitive information disrupting critical services, and in some cases, even directly destroying the physical devices or the environment [26, 35]. The primary cause of these cyberattacks can often be attributed to inherent vulnerabilities present in NEDs. Indeed, research indicates that approximately 70% of embedded devices possess exploitable vulnerabilities [55]. Consequently, it is critical for developers to prioritize vulnerability detection and mitigation during the implementation phase of NEDs.

Fuzzing is an exceptionally efficacious technique employed for the identification of security vulnerabilities, whereby it subjects the system under test (SUT) to unexpected and unpredictable input data [1, 63, 68]. In the specific context of NEDs, protocol fuzzing has gained prominence as a prevalent methodology for the evaluation of firmware security, with notable implementations including BooFuzz [6], PCFuzzer [43], AFLNET [57]. The goal of protocol fuzzing is to test communication interfaces of NEDs by generating a sequence of inputs, such as malformed packets or unexpected data values. Protocol fuzzing is distinguished by its ability to be fully automated and independent of access to source code. These characteristics make protocol fuzzing a scalable and efficient method for detecting vulnerabilities in NEDs [43].

However, fuzzing is a double-edged sword, and in particular, protocol fuzzing has significantly reduced the difficulty of finding vulnerabilities in **deployed NEDs**. Any malicious attacker can conveniently employ protocol fuzzing on an accessible NED to compromise the device through the exploitation of discerned vulnerabilities. To address this issue, vendors have implemented various approaches, encompassing vulnerability detection and mitigation techniques, to prevent attackers from exploiting vulnerabilities. While vulnerability detection during development is critical, it can not completely eliminate all vulnerabilities in NEDs. Moreover, the rectification of vulnerabilities after device deployment frequently often proves to be unfeasible and delayed. Empirical evidence substantiates this claim, revealing a prevalence of vulnerabilities within deployed NEDs, which surpasses the count encountered in conventional Information Technology (IT) devices [64]. Consequently, vulnerability mitigation techniques are preferred. Nevertheless, the implementation of mitigation techniques is limited due to the performance overhead of NEDs. Currently, vendors predominantly rely upon techniques such as code encryption and obfuscation to thwart static program analysis, while remaining vulnerable to dynamic analysis methodologies. This makes attackers favor low-cost protocol fuzzing instead of static analysis methods to discover vulnerabilities in NEDs. Thus, it becomes imperative to institute countermeasures against protocol fuzzing to protect the security of deployed NEDs.

Adversarial fuzzing techniques (*i.e.*, anti-fuzzing), ANTIFuzz [25] and FUZZIFICATION [29], have been proposed to disrupt the operational efficiency and effectiveness of fuzzing in traditional software, including command or file parsing programs. However, previous techniques can not be applied to NEDs against protocol fuzzing. The reason is that previous anti-fuzzing techniques primarily rely on intentionally slowing down the execution of SUTs by incorporating delays into the code of SUTs, particularly in the code snippets responsible for input reading or parsing. These techniques are designed for SUTs that read and process input *infrequently*, so the slowdown remains negligible compared to the overall execution time of SUTs. However, they incur significant time overhead to NEDs which *frequently* read and write data (*e.g.*, 1733x slowdown in our evaluation in §6.3.1). This consequential time overhead detrimentally impacts the availability and usability of deployed NEDs.

To design an effective anti-fuzzing technique for protocol fuzzing of NEDs, it is crucial to meet the following four requirements.

Efficacy It should significantly improve the complexity and difficulty encountered by protocol fuzzers in their quest to uncover vulnerabilities within NEDs.

Availability It should incur negligible time overhead to avoid affecting normal use of the protected NEDs.

Generality It should be generally applicable to various protocol fuzzing techniques, but not specific to a single protocol fuzzer.

Security It should not introduce new attack points or security weaknesses that could allow attackers to exploit and compromise the security of NEDs.

Simultaneously satisfying the above requirements is challenging (details in §2.3). This challenge stems from several intricacies. Firstly, the inherent randomness characteristic of fuzzing makes it difficult to identify specific features for effective countermeasures. Moreover, the fuzzer can dynamically generate test cases based on SUT feedback to cover more code. Secondly, the implementation of anti-fuzzing measures typically necessitates the incorporation of additional code and functionality. This means improved resource utilization, including device-sensitive time and space overhead. Thirdly, the landscape of fuzzer algorithms is marked by diversity and perpetual evolution, rendering the development of dedicated countermeasures tailored to each specific fuzzer impractical. Lastly, certain countermeasures that exhibit effectiveness may inadvertently introduce potential security risks. For instance, the adoption of measures such as imitating anti-debugging measures to terminate the program when subjected to fuzzing may inadvertently serve as potential security risks. This susceptibility arises as attackers could exploit this measure to execute denial-of-service (DoS) attacks on NEDs, where services operate as daemon processes, necessitating device restart for recovery. To the best of our knowledge, there is no prior technique that meets all four requirements.

Armor. This paper proposes the first anti-fuzzing technique named Armor, which is specifically designed to protect NEDs from protocol fuzzing. Concretely, we first design three adversarial primitives, which are delay, fake coverage, and forged exception. These primitives are specifically designed to counter the fundamental mechanisms that fuzzing relies on, which are high throughput of test execution, feedback-guided test case generation, and effective detection of exceptions in SUTs (details in §2.1). To avoid being exploited to attack NEDs, the adversarial primitives are specifically designed to limit their effects within a single network communication session without affecting the daemon communication process itself. Moreover, to ensure that the countermeasures do not affect the normal use of the protected NEDs, we design static and dynamic strategies to guide the use of adversarial primitives. The static strategy checks whether the message conforms to the protocol specification at the establishment phase of a session. If not, the static strategy enables adversarial primitives for the session to misguide fuzzing. On the other hand, the dynamic strategy inserts adversarial primitives in the code regions that are infrequently executed by normal use of NEDs. the dynamic strategy can complement the static strategy against advanced protocol fuzzers that leverage protocol-specific knowledge to help test sessions complete initial verification. Strategies are designed based on the observation (details in §2.3) that test cases generated during fuzzing usually violate the specification of the communication protocol and always execute the cold paths that normal inputs hardly ever execute Overall, Armor provides a comprehensive and effective solution to protect NEDs from protocol fuzzing while minimizing the impact on their normal use.

We conducted extensive evaluations of Armor. Based on four commonly used NED network protocols (Modbus-TCP [41], IEC 60870-5-104 [39], MQTT [50], and IEC 61850-MMS [40]), we evaluated the adversarial effectiveness of Armor against eight mainstream protocol fuzzers that cover all working principles (AFL [1], AFL++ [2], AFLNET [57], BooFuzz [6], KittyFuzzer [23], Peach [22], PCFuzzer [43] and SNIPUZZ [21]). The results show that Armor incurred negligible time overhead and effectively decreased the lines of code covered by fuzzing by 22%–61%, the number of discovered vulnerabilities by 57%–89%, and the throughput of fuzzers by more than 60%

when compared to the original (unprotected) program. Furthermore, when compared with the state-of-the-art anti-fuzzing techniques ANTIFuzz [25] and FUZZIFICATION [29], Armor performed better in terms of reducing code coverage and time overhead.

Contributions. We make the following contributions.

- We explored the challenges of applying the anti-fuzzing in NEDs, and designed mitigation techniques to reduce the likelihood of attackers discovering vulnerabilities in deployed NEDs through protocol fuzzing.
- We implemented the first technique Armor to protect NEDs from protocol fuzzing. We designed three adversarial primitives, *i.e.*, delay, fake coverage and forged exception, to counter the fundamental mechanisms of fuzzers in a general manner. Moreover, we designed static and dynamic strategies to arrange the adversarial primitives effectively and efficiently against protocol fuzzers.
- We evaluated Armor on four popular protocols used in NEDs and eight mainstream protocol fuzzers. The results demonstrate that Armor effectively reduces the line coverage of fuzzing by 22%–61% and the number of discovered vulnerabilities by 57%–89% with negligible time overhead.

2 BACKGROUND AND MOTIVATION

This section provides an overview of protocol fuzzing and its fundamental mechanisms. We also introduce prior anti-fuzzing techniques and explain our motivation and observations for designing Armor.

2.1 Protocol Fuzzing Techniques

Fuzzing is an automated software testing technique that generates a large number of test cases by either mutating existing test cases (known as seeds) [1, 43], or by generating new ones based on predefined templates [6, 23]. They all strive to generate various expected and unexpected inputs to rigorously test the SUT. Fuzzing technique involves running the SUT with the test cases and monitoring its runtime behaviors to collect feedback, such as code coverage and outputs, which guides the testing process and detects the manifestation of bugs. Protocol fuzzing is a special type of fuzzing technique that targets the testing of communication interfaces. Protocol fuzzing can be categorized based on how and where the firmware of NEDs is executed.

Emulation-Based Fuzzing. Fuzzing is executed through the emulation of device firmware within an emulator. In this category, the fuzzer, such as the AFL family [1, 57, 73], obtains code coverage information from the emulator to guide the fuzzing process. Nonetheless, the successful configuration of firmware within an emulator demands significant manual labor. At times, this task proves to be impossible due to the absence of essential software or hardware dependencies within the emulation environment. As a result, the application of this fuzzing technique on real NEDs is limited.

Device-Based Fuzzing. In this category, the test cases generated by a fuzzer are sent to the actual device through a network [6, 43]. This method is general and can be easily applied to various NEDs. However, it can only perform black-box testing due to the difficulty in collecting code coverage on devices. Black-box testing is usually less effective than coverage-guided testing [47]. To overcome this limitation, protocol fuzzers [21, 43] leverage the response messages from the device as a sub-optimal replacement of code coverage to guide the fuzzing process.

Despite the differences, the two categories complement each other in practice, and share the same fundamental mechanisms for effectively finding vulnerabilities, listed as below.

- High Throughput: The efficacy of fuzzing largely depends on the quantity of test cases that can be executed within a given time budget. Consequently, the attainment of a high throughput rate is crucial to optimize the probability of discovering vulnerabilities.

- Feedback-Guided Test Case Generation: Advanced fuzzers use feedback from the SUT to guide the generation of test cases. Feedback such as code coverage and response messages can help diversify the test cases generated to explore different behaviors of the SUT, leading to the discovery of previously unknown vulnerabilities.
- Effective Detection of Exceptions: Fuzzers rely on detecting abnormal states of the SUT to determine whether a test case triggers a vulnerability. This includes crashes and error semaphores, which must be monitored with a low false positive rate to ensure the effectiveness of the fuzzer. A low false positive rate is important to increase the trustworthiness and efficiency of the fuzzing process.

2.2 Prior Anti-Fuzzing Techniques

During the software development process, vendors typically employ various software testing techniques to find and eliminate bugs in programs. However, the deployed software usually still contains bugs that can be exploited by attackers hiding among users, just as Edsger Dijkstra said "Program testing can be used very effectively to show the presence of bugs but never to show their absence" [7]. To protect deployed software from vulnerabilities discovered by malicious fuzzing from potential attackers, anti-fuzzing techniques have been proposed. Vall-Nut [38] proposed three types of neutralizing schemes for queue explosion, seed attenuation and feedback contamination specifically for AFL fuzzer. ANTIFuzz [25] and FUZZIFICATION [29] techniques provided general anti-fuzzing solutions by adding delays and fake code paths to the input-checking and error-processing code regions. The target of protection of the above techniques is traditional software, such as command-line or file analysis programs. When these techniques are used to resist protocol fuzzing, they cannot be practically applied in NEDs. Table 1 provides a qualitative comparison of previous anti-fuzzing techniques for resisting protocol fuzzing in NEDs in terms of the following four dimensions.

- Efficacy: The above techniques resist traditional software fuzzing and thus do not take fuzzing test state into account. Traditional software fuzzing which completes a round of testing as soon as the input is obtained (*e.g.*. a command or a file). However, the protocol fuzzing involves the transfer of the session state machine, i.e. message ordering affects the parsing of the next message by the NED. Therefore, testers use prior knowledge to guide the test state with the correct message before the test case is sent, so that different code snippet have the opportunity to be tested, making the above techniques inefficacy [43].
- Availability: ANTIFuzz and FUZZIFICATION indiscriminately introduce supplementary codes into the input-checking sections to resist fuzzing. Therefore, they require significant execution overhead, particularly affecting the code paths responsible for input reading and parsing. When implemented in NEDs, characterized by frequent data read and write operations, these anti-fuzzing techniques precipitate a substantial deceleration in the execution pace of NEDs. This, in turn, significantly curtails the responsiveness of NEDs to normal user interactions, ultimately undermining the device availability.
- Generality: The mitigation strategies outlined by Vall-Nut pertain exclusively to the AFL fuzzing algorithm. Although there are many fuzzers based on AFL, there are also other fuzzers with different algorithms. As we mentioned in §2.1, there are various protocol fuzzers. Therefore, developing countermeasures specifically for specific fuzzers is not effective in mitigating the fuzzing risks encountered by NEDs.
- Security: The unacceptable processing overhead introduced by ANTIFuzz and FUZZIFICATION in normal communication consumes computing resources of NEDs, which can be exploited by attackers to launch resource consumption attacks. Moreover, FUZZIFICATION avoids detecting alerts by terminating the program when a fuzzer detects alerts without throwing exceptions. However, in NEDs, the service exists as a daemon process, and if a service terminates, the device needs to be restarted to recover. Attackers can use fuzzer or its results to launch denial of service attacks on the device.

Table 1. Comparison of resisting protocol fuzzing capabilities in NEDs. ●, ◐and ○represent support, partial support and no support, respectively.

Solutionss	Efficacy	Availability	Generality	Security
Vall-Nut	◐	●	○	●
ANTIFuzz	◐	○	●	○
FUZZIFICATION	◐	○	●	○
Armor	●	●	●	●

2.3 Motivation

Similar to traditional software, deployed NEDs also have vulnerabilities, and the situation is even more serious. On the one hand, resource limitations of NEDs and the lack of an update mechanism make it difficult to update software of NEDs, and the patching of vulnerabilities is not as timely and easy as traditional software [27, 54]. On the other hand, NED software development needs to consider more aspects, including not only API interface, data structure, compiler, but also hardware dependencies of peripheral drivers and integrated circuits. Therefore, there is a greater chance of bugs being created during the development of the firmware [30, 49]. To protect the security of deployed NEDs, vendors strip debug information and symbol tables from the released program, and use encoding, encryption, and obfuscation to thwart attackers. However, prior mitigation methodologies are only useful against static analysis techniques and have no countermeasure effect for protocol fuzzing. As a result, attackers tend to opt for low-cost protocol fuzzing to test deployed NEDs for exploitable vulnerabilities.

To effectively, practically protect NEDs from protocol fuzzing, we need to satisfy efficacy, availability, generality and security at the same time, as listed in Table 1. This task is non-trivial and mainly has the following ***challenges***. ① The first is to design a general confrontation technique that can counter the wide variety of off-the-shelf fuzzers and their different algorithms, including mutation and seed screening algorithms. This requires a deep understanding of the fundamental mechanisms of fuzzers and the ability to develop strategies that can effectively counter them. ② The second is to ensure that anti-fuzzing techniques do not introduce excessive time overhead, which could affect the availability of NEDs during communication. This is especially important since additional checking code needs to be inserted into the original program, which can increase the execution time. ③ The third challenge is that measures used to resist fuzzing may also introduce new attack points that can be exploited by attackers. For example, intercepting the abnormal exit signal of a protected program can cause the program to exit gracefully, avoiding abnormal monitoring. However, this can also be exploited to conduct denial of service attacks on NEDs. ④ Furthermore, protocol fuzzing involves multiple interactive tests in one session (not a single input but multiple ones to complete a round of testing), so attackers can combine different message sequences to assist the transition of the communication state machine to complete communication program test. This makes it possible for fixed countermeasures such as checking the legitimacy of certain messages to be bypassed, and the effectiveness of the independent confrontation mechanism for each test case is reduced.

The design of Armor is based on the following ***observations***. A network protocol specification defines the agreement on how messages should be transmitted between devices in a network, including the format of the messages, and how and when the messages should be sent and received in a session. In principle, this specification is not violated during normal communication. However, fuzzers attempt to generate diverse inputs to intensively test the SUT, and thus the fuzzer-generated inputs usually inconsistent with the protocol specification. For example, a handshake message is not sent at the very beginning of a session, or the magic byte in the message is incorrect. Therefore, most fuzzer-generated inputs cannot successfully establish a session, or they execute cold paths that normal communication processes rarely do. Based on the observations combined with the fundamental

mechanisms of protocol fuzzers in §2.1, we propose Armor mainly consisting of specifically designed adversarial primitives and adversarial strategies. The designed adversarial primitives, *i.e.*, delay, fake coverage and forged exception, are used to disrupt the fuzzing process. The static strategy and dynamic strategy are used to arrange the adversarial primitives according to the protocol specification and the cold path respectively, so as to decide whether to battle with the current session during the operation of NEDs. More details are in §4.

3 OVERVIEW OF ARMOR

This section introduces the application scenario and the threat model of Armor to understand its scope and capabilities. Moreover, the basic working principles are explained based on Armor's overall workflow.

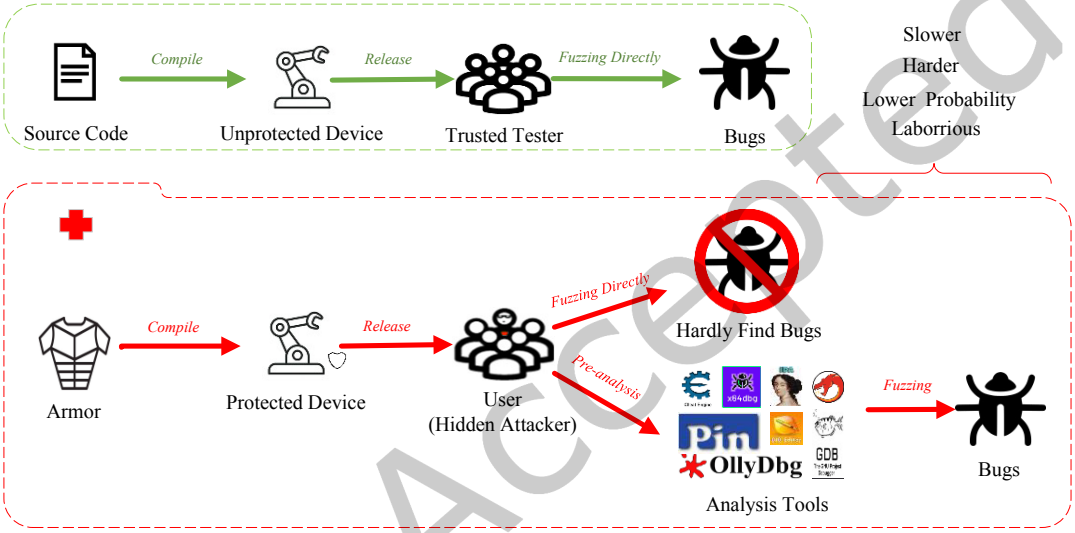


Fig. 1. Threat model.

3.1 Application Scenario

Armor is carefully designed to fortify NEDs against protocol fuzzing. The threat model shown in Figure 1 illustrates Armor's application scenarios and its ability to thwart malicious fuzzers. Specifically, the goal of Armor is to prevent attackers among users from using fuzzing to find vulnerabilities in NEDs by protecting the binaries executed on devices released to users. First of all, the protected device itself has the ability to resist fuzzing, so it is difficult to find bugs by directly applying fuzzing to the protected device. However, attack and defense are a process of evolving confrontation. Attackers in the real world can invest more resources and manual efforts to help carry out fuzzing, such as disassembly, emulation, and debugging. Note that increasing the attacker's analysis overhead and attack costs is also an effective defense measure [3, 33]. Moreover, combining anti-debugging, obfuscation, encryption and other protection measures can alleviate the above situation and further increase the cost of attacks (see §7 for detailed discussion). Therefore, it is at least slower, more difficult, less likely, and labor-intensive for untrusted users (hidden attackers) to discover vulnerabilities than when trusted testers fuzz on unprotected devices.

3.2 Overall Workflow

Based on the fundamental mechanisms of protocol fuzzers in §2.1 combined with the observations in §2.3, we design Armor as follows. At the beginning of the session establishment or in the cold path, Armor performs data checking to identify any data that is inconsistent with the protocol specification or executes the cold path. If such data is detected, Armor puts the current session into adversarial mode, which includes slowing down program execution speed to reduce fuzzer throughput, providing the fuzzer with fake test code coverage to misguide it into generating invalid subsequent inputs, and throwing forged exceptions to interfere with fuzzer results. This process ensures that the fuzzer diligently tests the adversarial code.

The workflow of Armor is shown in Figure 2 and Algorithm 1. We leverage the protocol library's testsuites to obtain execution porfiles of the library. Then static and dynamic strategies are implemented in the library according to the protocol specifications and porfiles to determine whether to launch the adversarial primitives. In this way, the client or server developed based on the protected protocol library has the ability to resist protocol fuzzing. Armor's three main functions are further explained below.

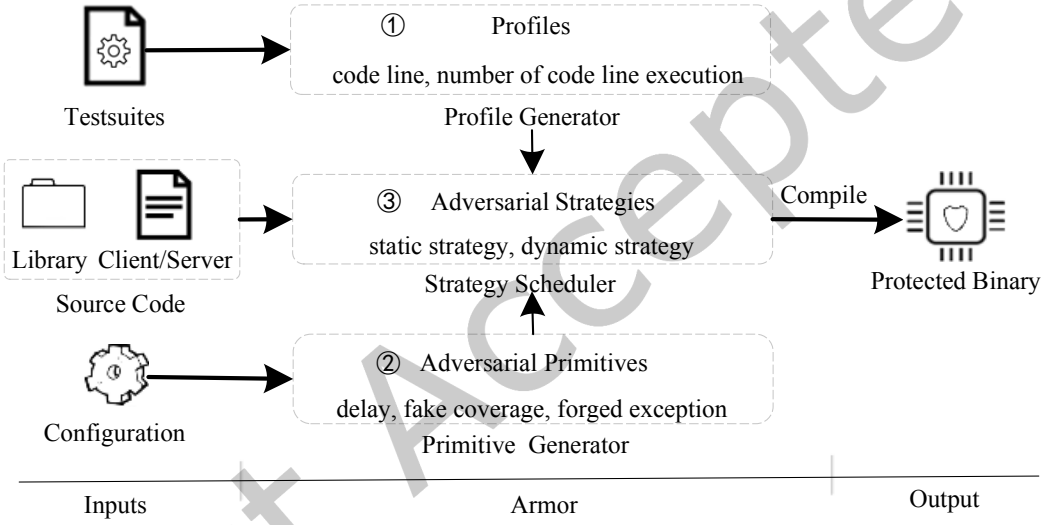


Fig. 2. Workflow of Armor.

① Profile Generator. We generate code-level frequency profiles for the protocol library. The profiles are then used to identify the cold path for injecting the adversarial primitives. The cold path is defined as the program path that is hardly or even never executed during normal communication. To reduce the reliance on manual operations for identifying the cold path, we use instrumentation technique with library testsuites to generate execution profiles for guidance. Specifically, commercial development projects often come with testsuites that cover code testing for functions in normal usage. We instrument and compile the code of the testsuites and corresponding library. Then we get the normal execution profiles of the library code by executing different testsuites. A profile contains code content and its execution times, which are provided to the strategy scheduler for analysis and use.

② Primitive Generator. We propose three adversarial primitives: *delay*, *fake coverage*, and *forged exception* to counter the fundamental mechanisms of protocol fuzzing. This ensures that Armor does not fail due to the emergence of new fuzzing techniques. To prevent the primitives from being easily removed, we design the primitives as callable functions that operate by receiving arguments (*i.e.* arg in Algorithm 1). In this way,

primitives have data and control dependencies with the code in the original program. Specifically, the delay primitive slows down program execution by performing intensive computations on arguments. The fake coverage primitive gives new feedback each time based on arguments, such as code coverage and response messages. The forged exception primitive emits non-reproducible exceptions that increase the false positive rate of the fuzzer. We provide these adversarial primitives to the strategy scheduler for use. These adversarial primitives are detailed in §4.1.

③ ***Strategy Scheduler.*** The static and dynamic strategies work together to decide the initiation of adversarial primitives. The *static strategy* performs specification consistency checks in the session establishment function to decide whether to apply adversarial primitives. For example, regular fields such as the constant magic bytes, fixed headers, and sequentially increasing sequence numbers in the protocol specification are checked. When checking for inconsistency, the input and subsequent inputs in the session are processed in adversarial mode. To prevent advanced attackers with protocol knowledge from helping the test complete session establishment, we design the *dynamic strategy*. According to the generated profiles the dynamic strategy inserts adversarial primitives and counters in cold paths for progressively enhanced adversarial. This attempts to minimize the impact of Armor on normal usage. The intuition is that normal users using legitimate clients or servers tend to provide valid inputs to NEDs, while fuzzers are more likely to trigger cold paths with invalid (malformed) inputs to NEDs. The adversarial strategies are detailed in §4.2.

Algorithm 1: Design of Armor.

Input: Library: Source code of library
Testsuites: Source code of testsuites
Program: Source code of client or server
Output: Protected binary
Data: arg: The received input data to process
// Adversarial primitive generation
delay(arg) ← Encryption calculation based on arg
fake coverage(arg) ← Call fake function or send fake response based on arg
forged exception(arg) ← Randomly generate forged exceptions, arg is used to generate data dependencies
// Static strategy implementation
protected receiver ← Insert consistency checks and primitives in message receiver function of Library
// Dynamic strategy implementation
profiles ← Compile, instrument and execute based on Library and Testsuites
cold paths ← Basic blocks not executed in profiles
protected cold paths ← Insert counters and primitives in cold paths of Library
Protected binary ← Compile Program with the protected Library
return Protected binary

4 PRIMITIVE AND STRATEGY DESIGN

The purpose of the adversarial primitives is to counter the fundamental mechanism of fuzzers and render their effectiveness futile. Nonetheless, to mitigate the potential adverse impact on normal users of NEDs, the adversarial strategy entails the systematic scheduling of adversarial primitives to determine their impact. This section provides a detailed exposition of the design aspects pertaining to adversarial primitives and adversarial strategies.

4.1 Adversarial Primitives

We design three adversarial primitives, namely delay, fake coverage, and forged exception to counter the three fundamental mechanisms of effective fuzzing, namely high throughput, feedback-guided test case generation, and effective detection of exceptions. This approach allows us to effectively combat fuzzing tests and ensures that our method remains effective even in the face of new fuzzers. To prevent attackers from easily removing the adversarial primitives, we design primitives with data and control dependencies between the primitives and the original code. The design details are as follows.

4.1.1 Delay Primitive. We first design delay primitive to break the high throughput mechanism of effective fuzzing. While using the `Sleep` function as the delay primitive is a possible solution, it can be easily detected and removed by attackers through reverse engineering. To avoid this, we utilize the implementation of encryption algorithms as the delay primitive, which is also time-consuming and can serve the purpose of delaying execution. Specifically, the design of the primitive involves wrapping the encryption algorithm in a loop, allowing us to control the specific delay time by modifying the loop times. Moreover, we use the received input as `arg` in the encryption algorithm to tightly couple the primitive to the original code of the protected program. This makes it extremely difficult for attackers to remove the delay primitive, ensuring its effectiveness. For security, we mainly compose the heavyweight computational tasks used to create the delay primitive of simple computations and repeatedly audited code such as encryption and hash calculation.

4.1.2 Fake Coverage Primitive. We design the fake coverage primitive to reduce the effectiveness of coverage-guided fuzzers in exploring the code space of the protected binary. Advanced fuzzers pick seeds by measuring the coverage of executing code in the emulation environment or the response message from NEDs. This primitive provides fake coverage feedback to inputs that do not have new coverage, causing fuzzers to consider these invalid inputs as seeds. This makes fuzzers prioritize the mutation of these seeds and waste significant time and computing resources exploring the primitive, resulting in meaningless fuzzing. To achieve this, we design two fake coverage primitives that target code coverage and response message coverage, respectively.

For fake code coverage primitive, the primitive prepares a large number of simple non-vulnerable functions (*i.e.*, fake functions) and form a call table. Upon receiving input, the primitive calculates the input's hash and selects a subset of fake functions based on this hash, which is then called in a specific order. This approach increases the data and control dependencies of the primitive with the original code, making it difficult for attackers to remove it. Furthermore, it prevents anomalies from being detected by fuzzers such as AFL, which can filter out unqualified seeds that produce different code coverage despite having the same content. Consequently, the primitive continuously generates new code coverage, including lines, functions, and branches, as newly generated inputs trigger the primitive.

For fake response coverage primitive, we construct a response message conforming to the protocol specification, but with some fields filled with random content. This intuition comes from the fact that black-box protocol fuzzers [21, 43] typically measure the response messages of test cases to infer whether the program has executed a new code path. If the device generates a new response message, the fuzzer continues testing with mutations based on the corresponding test case. By randomly filling some fields with content, the fuzzer that measures the response message can be tricked into selecting invalid seeds for testing. To ensure that the inserted code is associated with the original code, the primitive performs operations on inputs, such as comparisons. Furthermore, to avoid impacting normal users, the primitive is only triggered if the input violates the protocol specification (*i.e.*, static strategy).

4.1.3 Forged Exception Primitive. We design the forged exception primitive to interfere with fuzzing results by creating false positives. Hiding all abnormal behavior is an impractical solution because of the complex test environment and the variety of monitoring methods in NEDs. Moreover, some vulnerabilities directly lead to

program crash and are discovered. So we design forged exceptions instead of hiding anomalies to interfere with the test result. The rationale behind this is that the forged exceptions can give attackers confidence to continue testing, and also waste their time in reproducing and debugging false results. The previous approach [25] involved terminating the program to create a forged exception, but this approach is not suitable for NEDs that use a daemon process. If the program exits unexpectedly, it needs to restart the device to resume operation. To avoid being used by attackers to launch denial-of-service attacks on NEDs, we design forged exception primitive that only causes the current session to end rather than causes the program to terminate. Specifically, we forge four identical `SYN` messages to obtain the correct session sequence number, and then forge a `RST` message to end the session without killing the process. At the same time, we forge an exception at this point by adding a long delay. In this way, both TCP-based and UDP-based protocols can perform exception interference. Similar to the previous primitives, the inserted code is associated with the original code through operations such as comparisons on the input.

4.2 Adversarial Strategies

To effectively counter fuzzing while ensuring program availability, we propose static and dynamic strategies. These strategies provide guidance on where to implement and when to initiate adversarial primitives. The design intuition is that fuzzing usually generates test cases that violate protocol specifications, including syntax and semantics. Moreover, these malformed test cases tend to execute cold paths, which are code regions that would rarely be executed by normal inputs. Building on this observation, we design static and dynamic strategies to combat fuzzing, which are detailed below.

4.2.1 Static Strategy. In the static strategy, we begin by examining whether the input received during session establishment is from users or fuzzers. When the input is detected as being generated by fuzzers, we apply adversarial primitives. Advanced fuzzers make an effort to generate varied inputs to rigorously test the SUT, often leading to inputs or input sequences that violate the protocol specification. For example, the magic byte may be improperly assigned without calculation, or the sequence numbers may be arranged in an incorrect order. Therefore, we determine whether the input is from fuzzers by examining the fixed offset of the message or the fixed sequence of messages. The header field of the Modbus-TCP protocol as shown in Figure 3. The Modbus-TCP protocol specification [41] stipulates that the protocol identifier is two bytes of `0x00`. We can add checks for bytes with input offsets 2 and 3 at the message receiving function (_modbus_tcp_recv) in the library. When the content of these two bytes violates the specification, the adversarial primitives take part in the processing of input and subsequent input in the session. This strategy is generally sufficient for most fuzzers without expert guidance since the testing process can be misguided by adversarial primitives to perform invalid tests at the beginning of fuzzing. For state-based protocol fuzzers that utilize legitimate messages to guide session processes initially, the dynamic strategy detailed below can combat them.

Transaction Identifier	Protocol Identifier	Length Field	Unit ID
(2 Bytes)	(2 Bytes)	(2 Bytes)	(1 Bytes)

Fig. 3. Modbus-TCP application protocol header.

4.2.2 Dynamic Strategy. In the dynamic strategy, we check whether the protected program executes to cold paths, and then apply adversarial primitives when the cold paths are executed. Specifically, to identify cold paths, we use the instrumentation technique and run all accompanying testsuites to generate profiles that record the number of code executions. Code lines in the library that have zero execution are considered cold paths.

For example, some exception handling snippets often belong to cold paths, because normal messages from the client or server that comply with the protocol specification will not cause processing exceptions. Then, we apply the adversarial primitives to cold paths, but unlike the static strategy, the effect of the primitives is gradually enhanced. We do this for two reasons. First, protocol fuzzing tests often require multiple interactions within a single session, and gradual enhancement of adversarial primitives proves more effective in such scenarios. Second, most of the test cases from the fuzzers do not satisfy the protocol specification and have a higher probability of executing to cold paths. However, normal inputs may occasionally execute on cold paths due to network quality. Therefore, we cannot directly apply heavyweight primitives on cold paths, which may prevent normal communication. To this end, we gradually increase the effect of the adversarial primitives based on the number of times the cold path is executed in a session. The intuition is that the more times the cold path is executed in a session, the higher the probability that the input to the current session is generated by fuzzers. We do this by adjusting the delay time and the probability of activating forged exceptions, using the calculation equation $Result = w * counter * initial\_value$, where w is the weight, $counter$ refers to the times of traversing the cold paths in a session, and $initial\_value$ denotes the initial delay time or activation probability of forged exceptions.

5 IMPLEMENTATION

We implemented the prototype of Armor using Python and C. The implementation details of the modules of Armor are as follows. The *profile generator module* was implemented using the Clang/LLVM compiler. We instrumented the source code to obtain information about the execution frequencies of library files when run with testsuites. For the *primitive generation module*, we used the ANTIFuzz framework [25] and made modifications to support the generation of the fake response coverage primitive and forged exception primitive. The *strategy scheduler module* was implemented in Python, including the static and dynamic strategies. The static strategy deploys primitives at message receiving functions in the library. The dynamic strategy deploys primitives based on the generated profiles. Once the strategies have been deployed, we can compile the source code of the client or server to be protected, resulting in a program with the ability to resist protocol fuzzing. Moreover, Armor allows for parameter configuration to better meet the needs of developers. The configuration file can be modified to adjust parameters such as the delay time, the number of fake paths, and the throwing probability of forged exceptions, as well as the content of static strategy checks and the weighting of dynamic strategy.

6 EVALUATION

We designed the evaluations of Armor to answer the four research questions below:
RQ1: Efficacy and generality of Armor against protocol fuzzing.
RQ2: Availability of NEDs after applying Armor.
RQ3: Security of NEDs after applying Armor.
RQ4: Effect of Armor parameters on protection and overhead.

6.1 Evaluation Setup

Evaluated Protocols. As shown in Table 2, we use four protocols widely used in NED work scenarios for Armor evaluation. The servers in the library testsuites corresponding to each protocol are our test objects. Modbus-TCP [41] is a standard communication protocol for connecting industrial electronic devices. IEC 60870-5-104 [39] is an international standard for data exchange between control stations and substations in electrical power systems. MQTT [50] is designed for connections with remote locations that have devices with resource constraints or limited network bandwidth. IEC 61850-MMS [40] has been successfully used in many commercial software products and embedded device in smart substation. These four protocols are widely used in security analysis studies of NEDs [10, 16, 34, 45, 59, 72, 74].

Table 2. Details of Evaluated Protocols

Protocol	Library version	SUT
Modbus-TCP	3.1.6	random-test-server
IEC 60870-5-104	2.3.0	simple-server
MQTT	2.0.11	mosquitto-broker
IEC 61850-MMS	1.5.0	server_example_basic_io

Evaluation Metrics and Baselines. To answer the proposed research questions, we select different metrics and baselines as follows.

- **RQ1**: To illustrate efficacy and generality of Armor, we selected eight typical ***protocol fuzzers***: ① AFL [1], emulation-based fuzzer guided by code coverage, works with the desock module of Preeny [58] in QEMU mode (since the source code of software in commercial NEDs is usually not available, we use QEMU mode to test). We also chose ② AFL++ [2] in QEMU mode, an improved version of AFL, and ③ AFLNET [57], which is specialized for protocol fuzzing. ④ BooFuzz [6], ⑤ KittyFuzzer [23], and ⑥ Peach [22] are generation-based protocol fuzzer. ⑦ PCFuzzer [43] and ⑧ SNIPUZZ [21] are mutation-based protocol fuzzer. We used the following ***metrics*** to evaluation: ❶ code coverage and ❷ the number of discovered vulnerabilities by following the fuzzing evaluation recommendations [32], and also used ❸ throughput which is a measurement for fuzzers [73]. We chose three ***baselines*** for comparison: the unprotected original program and programs protected by the state-of-the-art anti-fuzzing techniques, *i.e.*, ANTIFuzz [25] and FUZZIFICATION [29]. We do not compare with Vall-Nut [38] because it is not open source.
- **RQ2**: To illustrate availability of the protected binary, we employed time overhead and program size overhead as quantifiable ***metrics***. These metrics allow us to assess the practical accessibility of the protected binary. Similar to RQ1, we selected the unprotected original program, as well as the programs secured by ANTIFuzz and FUZZIFICATION techniques, as ***baseline*** references for comparison.
- **RQ3**: To illustrate that there is no new security risks after applying Armor, we analyzed other session states of the protected binary under fuzzing and used the alert information recorded by fuzzers to verify the vulnerability of the protected binary. The evaluation ***metrics*** include that the impact of Armor is limited to the fuzzing session, and forged exceptions cannot be reproduced. We chose ANTIFuzz which also has exception interference function as the ***baseline***.
- **RQ4**: To illustrate the impact of Armor's parameters on its effectiveness in providing protection and the associated overhead costs, we conducted a series of comparative experiments. These experiments entailed the exploration of diverse settings for Armor's parameters, encompassing factors such as delay time, the count of fake paths, the probability of forged exceptions, and the weight of dynamic strategy. To assess the outcomes of these experiments, we used the number of discovered bugs, time overhead, and size overhead as ***metrics***.

Armor Configuration. Armor is designed to be used by developers. who can configure its parameters based on the attributes of the protocol used by the protected program, such as interaction frequency and real-time requirements. For the purposes of evaluation, the protocols tested in this evaluation were all configured with the following parameters: the initial delay time of 1 second, the generation of 3000 fake paths, the forged exceptions triggering frequency of 1/30, and the weight parameter values of the delay primitive and forged exception

primitive in the dynamic strategy are 1 and 0.1, respectively. These values were determined through conservative experimentation with the evaluated protocols.

Hardware. Our experiments were performed on a Linux workstation with an Intel Core i7-8750H CPU and 64G RAM.

6.2 Efficacy and Generality of Armor

6.2.1 Reducing code coverage. We used the real code coverage of the SUT to evaluate efficacy of Armor. Real code coverage refers to the original code of the protected program, excluding the code of the primitives, because our strategy is to induce the fuzzer to waste time in testing code of the primitives. To obtain real code coverage for fuzzers, we employed the instrumentation technique when compiling the source code to binary. Note that instrumentation information is not included in the compiled binary to be released.

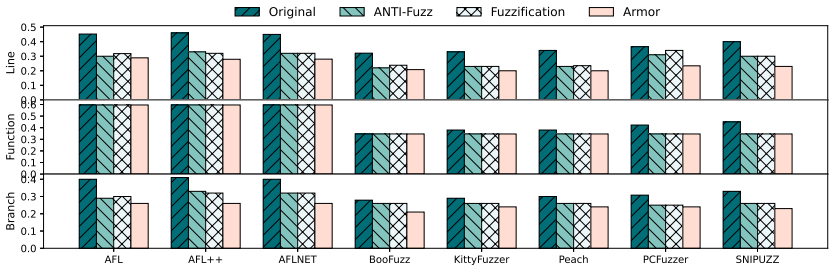
We leveraged three popular code coverage metrics for evaluation: line coverage, function coverage, and branch coverage. Figure 4 shows the code coverage results in Modbus-TCP, IEC 60870-5-104, MQTT, and IEC 61850-MMS servers. Here we took the average results of three 24-hour fuzzing rounds. From the results, we find that the fine-grained code coverage (*i.e.*, line coverage and branch coverage) of the protected binary of Armor is significantly reduced. For example, the line coverage is 22%–61% lower than the original binary. Compared with ANTIFuzz and FUZZIFICATION, Armor reduces the line coverage by 3.7%–44.4%. Note that we also conducted longer fuzzing lasting 72 hours, but no noticeable change in code coverage was observed. This is because the fuzzers have been misled to make them difficult to generate valid test cases. To further illustrate the difference of test cases generated for protected and unprotected programs, we used the Mann Whitney U test to determine statistical significance [32, 38], which provides a measure of the degree to which a pair of sets differ. A `p-value` smaller than 0.05 means the difference between the two sets is statistically significant. Table 3 shows that all the p-values between the protected program and the original program are smaller than 0.05. This indicates that Armor can significantly reduce the code coverage of the protocol fuzzers from the statistics perspective.

Table 3. Significance of evaluations of fuzzer pairs using p-value from the Mann-Whitney U-Test.

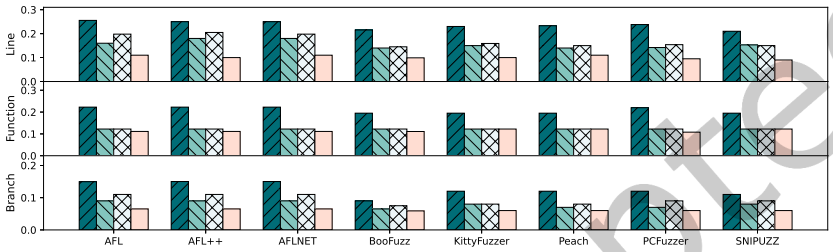
	AFL	AFL++	AFLNET	BooFuzz	KittyFuzzer	Peach	PCFuzzer	SNIPUZZ
Modbus-TCP	$5*10^{-4}$	$5*10^{-4}$	$8*10^{-5}$	$2*10^{-3}$	$4*10^{-3}$	$3*10^{-3}$	$1*10^{-4}$	$5*10^{-4}$
IEC 60870-5-104	$9*10^{-5}$	$2*10^{-4}$	$3*10^{-5}$	$6*10^{-3}$	$9*10^{-3}$	$8*10^{-3}$	$5*10^{-5}$	$3*10^{-4}$
MQTT	$8*10^{-4}$	$9*10^{-5}$	$2*10^{-5}$	$5*10^{-3}$	$8*10^{-3}$	$5*10^{-3}$	$2*10^{-3}$	$7*10^{-3}$
IEC 61850-MMS	$5*10^{-4}$	$5*10^{-4}$	$6*10^{-5}$	$2*10^{-3}$	$5*10^{-3}$	$5*10^{-3}$	$2*10^{-4}$	$7*10^{-4}$

6.2.2 Hindering vulnerability finding. We evaluated the impact of Armor on vulnerability finding by measuring the number of vulnerabilities discovered by fuzzers on both the protected and original binaries. We did not compare our results with binaries protected by ANTIFuzz and FUZZIFICATION due to their heavy adversarial techniques that cause the protected binaries to deviate from normal execution, which breaches the availability requirement of anti-fuzzing for NEDs. Even normal messages may cause the session to be terminated due to timeout.

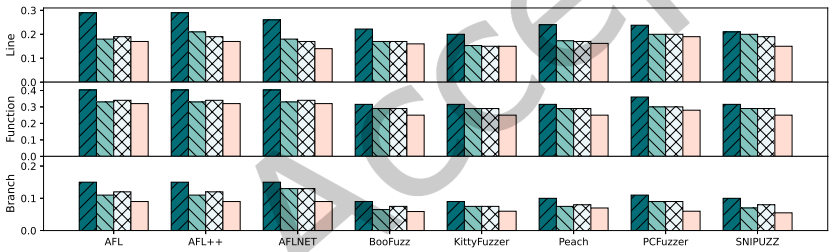
To evaluate Armor's ability to hinder vulnerability finding, we randomly inserted 20 real crashes into the evaluated program, as vulnerabilities are mainly introduced into the program due to secondary development of the protocol library by developers. We conducted three rounds of 72-hour fuzzing to ensure consistency in the evaluation. Fuzzers recorded all exceptions, including those generated by the forged exception primitive, but after verification, we found that forged exceptions cannot be reproduced. Therefore, we only counted the number



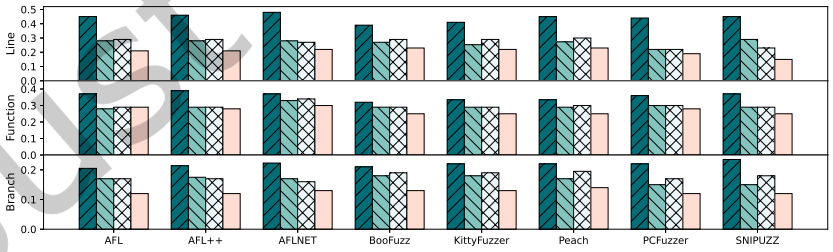
(a) Code coverage of Modbus-TCP.



(b) Code coverage of IEC 60870-5-104.



(c) Code coverage of MQTT.



(d) Code coverage of IEC 61850-MMS.

Fig. 4. Test code coverage of original and protected programs by different protocol fuzzers. Lower code coverage means a lower likelihood of finding vulnerabilities. The ordinates represent different code coverages, and the abscissas represent different fuzzers.

of real unique vulnerabilities found by the fuzzers. Table 4 shows that Armor reduces the number of discovered vulnerabilities by 57%–89% compared to the original binaries. This reduction is primarily due to the reduced code coverage of fuzzer tests induced by our protection strategy.

Table 4. Number of vulnerabilities found.

	AFL		AFL++		AFLNET		BooFuzz		KittyFuzzer		Peach		PCFuzzer		SNIPUZZ	
	Orig.	Armor	Orig.	Armor	Orig.	Armor	Orig.	Armor	Orig.	Armor	Orig.	Armor	Orig.	Armor	Orig.	Armor
Modbus-TCP	17	3	17	4	15	2	5	1	8	1	8	1	9	2	11	2
IEC 60870-5-104	18	2	18	2	17	3	7	2	5	1	6	1	9	2	9	3
MQTT	18	4	18	4	18	3	7	3	7	1	9	2	10	3	7	1
IEC 61850-MMS	16	3	16	3	17	3	8	3	8	2	10	2	10	2	11	2

6.2.3 Reducing throughput. We measure the ability of Armor to reduce throughput, which is the rate at which test cases are executed during fuzzing. Slowing down the program under fuzzing causes the attacker to spend more time completing the same number of test cases. To evaluate this, we compare the throughput of all fuzzers when applied to both the protected binary and the original binary. We calculate the ratio of average value of instantaneous throughput and present the results in Table 5. The results show that Armor reduces the throughput of all fuzzers by more than 60%. To further intuitively illustrate Armor's effectiveness in deferring potential attackers, the time overhead of sending the same test cases (10,000) is shown in Table 6. A potential attacker would need to spend at least 2.5x longer to complete the same number of test cases. The reduction in throughput can be attributed to two factors. On the one hand, the adversarial primitive implemented in Armor slows down the execution rate of the protected program under fuzzing. On the other hand, the adversarial primitive increases the computational complexity of the fuzzer itself. While ANTIFuzz and FUZZIFICATION reduce throughput even more than Armor, their heavyweight adversarial techniques violate the availability requirement of anti-fuzzing for NEDs, making both techniques unsuitable for protecting NEDs.

Table 5. The degree of decrease in fuzzing throughput, calculated by `(orig-Armor)/orig`.

	AFL	AFL++	AFLNET	BooFuzz	KittyFuzzer	Peach	PCFuzzer	SNIPUZZ
Modbus-TCP	98%	97%	99%	75%	71%	72%	63%	62%
IEC 60870-5-104	66%	66%	69%	64%	63%	66%	62%	60%
MQTT	72%	70%	78%	65%	66%	65%	61%	61%
IEC 61850-MMS	69%	68%	69%	64%	66%	66%	68%	60%

Table 6. Time overhead for sending the same number of test cases, calculated by `Armor/orig`.

	AFL	AFL++	AFLNET	BooFuzz	KittyFuzzer	Peach	PCFuzzer	SNIPUZZ
Modbus-TCP	49.1x	44x	83.3x	4.1x	3.6x	3.6x	3.7x	2.8x
IEC 60870-5-104	3.4x	3.2x	3.6x	2.9x	3.9x	3x	2.7x	2.5x
MQTT	3.6x	3.3x	4.2x	2.7x	2.9x	2.9x	2.7x	2.7x
IEC 61850-MMS	3.3x	3.2x	3.4x	2.8x	2.9x	2.9x	3.1x	2.5x

Answer to RQ1: Armor has been shown to be effective against eight mainstream protocol fuzzers, demonstrating its efficacy and generality. In comparison with the original program, Armor reduces line coverage by 22%-61%, outperforming ANTIFuzz and FUZZIFICATION. Furthermore, Armor decreases the number of discovered vulnerabilities by 57%–89% and reduces fuzzing throughput by over 60%.

6.3 Availability of NEDs

6.3.1 Time overhead. For embedded devices, anti-fuzzing techniques should not affect normal communication. To assess this aspect, we conducted evaluations by monitoring the message processing time exhibited by the binary protected with the anti-fuzzing technique. We compared this observed processing time with that of the original binary. Figure 5 shows the average time spent per message for ten trials. The communication time fluctuates slightly (millisecond level) due to network and hardware quality, so the ten rounds of trials are represented in the form of box plots. The evaluation shows that the time overhead of applying Armor is negligible, and the times of Armor-protected binaries (e.g. Modbus-TCP) are even smaller than the worst-case times of the original binaries due to communication fluctuations. These mean that Armor does not affect normal communication for NEDs. This is a crucial requirement for anti-fuzzing techniques applied to NEDs because they must maintain availability during normal operation. In contrast, the time overhead of applying ANTIFuzz and FUZZIFICATION is significant, ranging from 50–160 milliseconds per message, which can result in a large increase in the time required for normal communication due to frequent message interactions. For example, the binary protected by ANTIFuzz took 156,000 milliseconds to complete the entire test in Modbus random test server, which was 1733X slower than the original binary. The large time overhead of ANTIFuzz and FUZZIFICATION is due to the fact that they take countermeasures for each input, resulting in a significant increase in processing time for programs that communicate frequently.

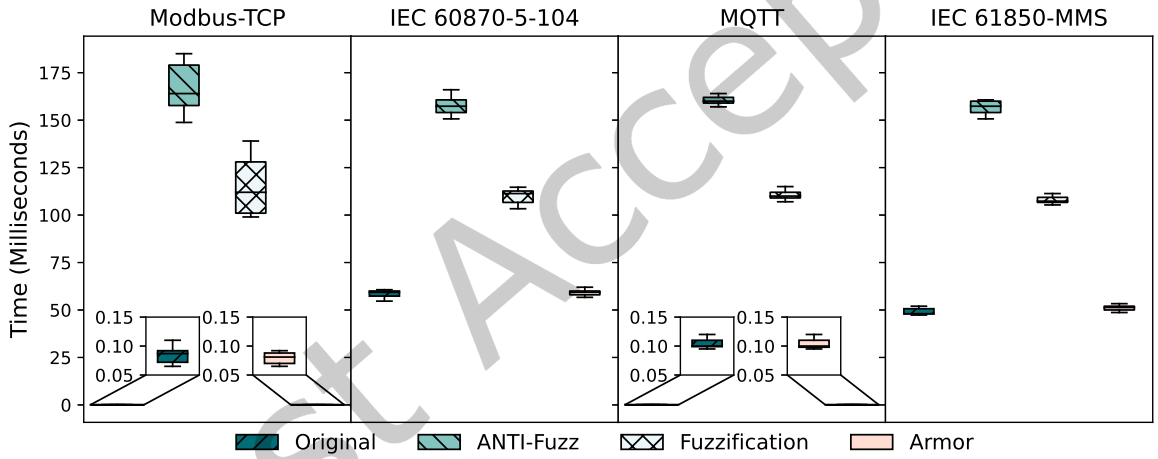


Fig. 5. Time spent per message.

6.3.2 Size overhead. Considering the limited memory resources of embedded devices, it is important to ensure that the size of the protected binary remains within manageable bounds. In response to this concern, we conducted a detailed analysis of the supplementary code generated by Armor. The size overhead incurred by the current Armor configuration is almost constant at approximately 4.6MB. The factor that affects the size overhead is mainly the number of fake paths (see §6.5 for details). In comparison, the additional code generated by ANTIFuzz and FUZZIFICATION stood at around 25MB and 2MB, respectively. While Armor does entail a slightly greater size overhead compared to FUZZIFICATION, our investigation suggests that these overheads remain well within acceptable limits. This is founded on the consideration that prevalent NED firmware typically falls within the size range of 30MB to 150MB. Moreover, the memory capacity of devices tasked with storing firmware typically ranges from 256MB to 512MB. Therefore, the size overhead incurred by Armor is deemed negligible for NEDs.

Answer to RQ2: The application of Armor does not compromise the availability of NEDs. In comparison to the original program, the implementation of advanced anti-fuzzing techniques such as ANTIFuzz and FUZZIFICATION result in time overheads that are at least 600 times greater, whereas Armor incurs almost no time overhead. Furthermore, the size overhead associated with Armor is deemed acceptable.

6.4 Security of NEDs

To ensure the security of NEDs, any added functionality from anti-fuzzing techniques must not introduce potential risks for attacks. Therefore, it is important to limit the effect of adversarial primitives to the session under fuzzing. If these primitives can affect the communication program itself by causing the program to terminate, fail to establish a connection, or interrupt other sessions, the primitives can be exploited as attack methods. To address this, we conducted an analysis to ensure that the effect of primitives is limited to the session under fuzzing and does not impact other sessions. Specifically, we established a session with a running server program and performed fuzzing while simultaneously establishing another session with the same server for normal communication. As shown in Table 7, our results demonstrate that none of the normal sessions for the four protocols were affected by the fuzzing session. Therefore, we confirm that the effect of the three primitives is confined to the session under fuzzing and does not impact the program itself, ensuring the security of NEDs. In contrast, programs protected by ANTIFuzz and FUZZIFICATION as evaluated in §6.3.1 above cannot even communicate normally. Moreover, when fuzzing is running, the program protected by ANTIFuzz terminates the process to create an exception, which adversaries can exploit to launch a denial of service attack against deployed NEDs.

Table 7. Whether normal communication sessions are affected.

	ANTIFuzz	FUZZIFICATION	Armor
Unacceptable delay	✓	✓	×
Denial of service	×	✓	×

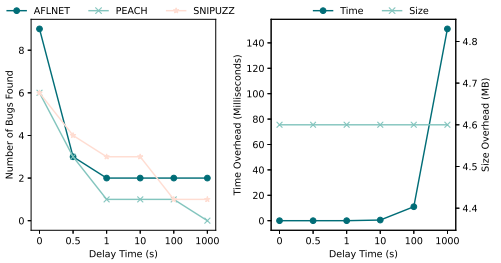
We further evaluate the results of the fuzzing, because Armor additionally introduces forged exceptions. For actual exceptions, Armor reduces the number and likelihood of bug finding §6.2.2. For forged exceptions, they cannot have any effect on the program itself. We analyzed the fuzzing outcomes caused by forged exceptions, as well as actual exceptions. While actual exceptions triggered segmentation faults and stack overflows, the verification data causing forged exceptions did not result in program faults or overflows. Therefore, we conclude that these forged exceptions cannot be reproduced, and forged exception primitives do not become new attack points for NEDs. Forged exceptions only interfere with test results and waste attacker analysis time.

Answer to RQ3: The functionality of Armor is security and its impact is limited to a single network communication session without affecting the program. Therefore, it is not feasible for attackers to exploit the features of Armor to initiate an attack against the protected NEDs.

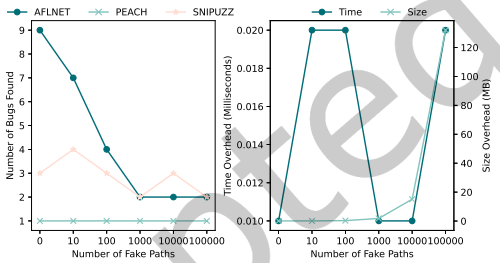
6.5 Effect of Parameters

We measure the effect of different parameter settings of Armor on the protection effectiveness and overhead cost of adversarial fuzzing. This provides developers with guidance in selecting parameters based on their preference between protection level and overhead. Specifically, we analyze the impact of the five parameters, including the delay time, the number of fake paths, the probability of forged exceptions, and the two weights of dynamic

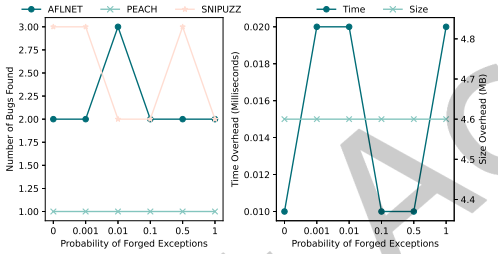
strategy. To evaluate protection effectiveness, we employ the metric of the number of real bugs detected by fuzzers during a 72-hour time frame. Our assessment involves the utilization of well-performing fuzzers from the categories of coverage feedback-based, template generation-based, and response feedback-based fuzzers in §6.2, namely AFLNET, Peach, and SNIPUZZ, respectively. For the assessment of overhead cost, we consider both time overhead and size overhead as metrics. We make variations to one parameter at a time while keeping the remaining parameters aligned with the configuration outlined in §6.1. Figure 6 presents the evaluation results based on Modbus-TCP. For the other three protocols IEC 60870-5-104, MQTT and IEC 61850-MMS, the impact of parameters exhibit similar trends as Modbus-TCP. We analyze the results in detail as follows.



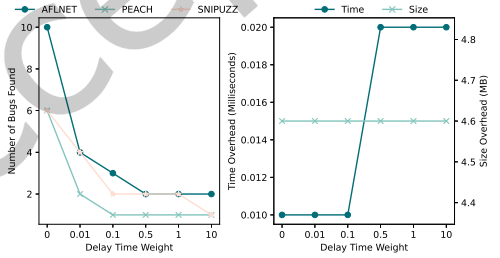
(a) The impact of delay time.



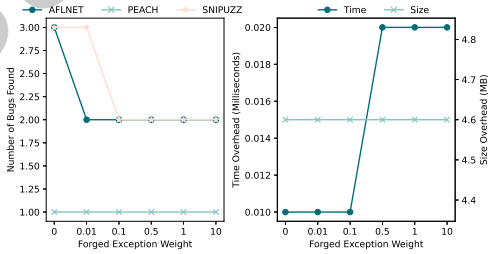
(b) The impact of the number of fake paths.



(c) The impact of the probability of forged exceptions.



(d) The impact of the weight of delay in dynamic strategy.



(e) The impact of the weight of forged exception in dynamic strategy.

Fig. 6. The impact of Armor parameters on protection effectiveness and overhead costs based on Modbus-TCP.

Figure 6a: As the delay time increases, the throughput of the fuzzer decreases, resulting in a decrease in the number of bugs found by the fuzzer. Moreover, delay time has a significant impact on time overhead but has little impact on size overhead.
Figure 6b: The increase in the number of fake paths leads to a decrease in the number of bugs discovered by the fuzzer based on coverage feedback. But it has little impact on other types of fuzzers. Furthermore, the program size overhead is sensitive to the number of fake paths, but the time overhead is not sensitive and fluctuates within a small range (about 0.01 milliseconds).
Figure 6c: The probability of forged exceptions only affect the number of forged exceptions in the test results, thereby wasting bug verification time. There is little impact on the real number of bugs, time overhead, and size overhead. It is worth mentioning that if the probability is greater than or equal to 1, each malformed message will trigger session abort.
Figure 6d The overall trend of the weight of the delay time in the dynamic strategy is similar to the delay time in Figure 6a. The difference is that the time overhead caused by weight value is significantly reduced. However, high weight value also have side effects, which can aggravate the impact of abnormal message on sessions due to communication quality or operational errors.
Figure 6e The weight of forged exception in the dynamic strategy has no obvious impact on the number of bugs and overhead, which is similar to Figure 6c. The high weight value may result in frequently aborted sessions due to malformed messages.

Answer to RQ4: The parameter configuration of Armor has varying degrees of impact on protection effectiveness and overhead costs. Parameter adjustments are made based on specific overhead tolerance and the trend of influence of each parameter.

7 DISCUSSION

To gain a deeper comprehension of Armor's scope and capabilities, the following section discuss the purpose of offensive and defensive confrontations and the limitations of Armor.

7.1 Purpose of Anti-fuzzing

The purpose of Armor is not to hide all vulnerabilities from potential attackers, but rather to increase the cost of discovering vulnerabilities through fuzzing, which is in line with other attack mitigation measures. For instance, vendors often utilize reverse-engineering mitigation measures like code obfuscation and encryption to prevent attackers from performing static analysis on released software. However, attackers can still crack and analyze the released software through de-obfuscation techniques [46], brute force cracking [31], and information leakage [69]. Moreover, to prevent dynamic analysis such as debugging, dynamic instrumentation, and virtual execution, vendors usually use detection flags [67], self-detection [42], runtime checks [4], *etc.*, to combat them. However, these countermeasures can also be bypassed through techniques such as code patching [17] and rewriting [8]. Similarly, vulnerability exploitation mitigation measures such as Address Space Layout Randomization (ASLR), No-EXecute (NX), and Canary are designed to prevent further exploitation of vulnerabilities once discovered. However, attackers can circumvent or disable these mitigation measures through side channels [19, 28] or design flaws [9] . Like these mitigation measures, Armor cannot provide complete defense against attacks, but rather defends by increasing the attack's cost. Additionally, vendors and trusted third parties, who conduct fuzzing on unprotected devices, have a greater chance of discovering and patching vulnerabilities in a timely manner than attackers who test on the protected devices.

7.2 Anti-anti-fuzzing

The game of attack and defense has always been an ongoing concern in the realm of cybersecurity. As fuzzing techniques continue to evolve, corresponding anti-fuzzing techniques must also be developed. With this in mind, we designed Armor to anticipate potential future against anti-fuzzing techniques (*i.e.*, anti-anti-fuzzing). First, attackers can attempt to reverse engineer the protected binary to remove the adversarial code. To this end, the adversarial primitives of Armor have data and control dependencies with the original code of the target program, making it challenging for attackers to remove them. Moreover, we can also use code obfuscation and encryption to further protect binaries from reverse engineering. Second, attackers can invest more effort or even manually to design new and effective test cases. However, creating effective test cases is a non-trivial task that requires a deep understanding of the binary and sophisticated program analysis techniques. In this case, Armor breaks the highly automated nature of fuzzing and increases the cost of attack.

7.3 Performance Overhead

Like other attack mitigation measures, such as ASLR, Armor enhances the security of NEDs at the cost of performance overhead. Compared to traditional computer software, performance overhead, particularly execution time overhead, can have a greater impact on NEDs since they are used in scenarios such as industrial control and smart cars, where high real-time requirements exist. As noted in §6.3, we have observed that Armor incurs almost no additional time overhead compared to the original program, which is also superior to the state-of-the-art work ANTIFuzz [25] and FUZZIFICATION [29]. The program size overhead of Armor is inevitable. However, the resulting size overhead is acceptable in NEDs, and we provide configurable parameters for the generation of primitives, with the number of fake paths being the primary factor affecting the size overhead. Based on actual needs, developers can make an informed decision regarding the best trade-off between security and performance. Furthermore, Armor implements protection strategies in the library, which makes applications or executable binaries developed based on the library have the ability to resist protocol fuzzing. Armor's ideas can also be migrated to the full program through case-by-case instrumentation and code analysis. This will also bring about improvements in protection effectiveness and performance overhead.

8 RELATED WORK

In this section, we present an overview of related work, including anti-fuzzing techniques, protocol fuzzing techniques, and other anti-analysis techniques for vulnerability analysis.

8.1 Anti-Fuzzing Techniques

As the number of vulnerabilities discovered through fuzzing continues to increase, researchers have begun exploring anti-fuzzing techniques. One such technique, proposed by Edholm *et al.* [18], involves concealing the feedback signal when the software behaves abnormally, thereby preventing the fuzzer from monitoring the abnormality. However, there are various ways to detect anomalies, such as through binary dynamic instrumentation, virtual machine debugging information, and program output messages. Moreover, not all anomalies can be concealed, such as program output messages and program crash exits [43]. Therefore, this anti-fuzzing method is easily bypassed by attackers. To reduce the speed of program execution and resist fuzzing, ANTIFuzz [25] and FUZZIFICATION [29] insert additional delay and false code on the path of input reading and parsing. However, these methods are only applicable to programs that read data infrequently and can cause availability issues when applied to NEDs. Vall-Nut [38] designed neutralizing mechanisms to undermine the effectiveness of fuzzing by studying the seed evaluator and mutator of AFL. However, this method is only valid for the AFL family of fuzzers and not for other types of fuzzers such as black-box fuzzers. Armor, on the other hand, combines adversarial

primitives and strategies to achieve efficacy, availability, generality, and security against protocol fuzzing for NEDs.

8.2 Protocol Fuzzing Techniques

Protocol fuzzing does not require access to the source code and can be fully automated, making it a highly efficient and scalable approach to vulnerability detection in NEDs. Protocol fuzzing can be categorized into emulation-based fuzzing and device-based fuzzing according to how and where the firmware of NEDs is executed.

For emulation-based fuzzing, analysts emulate the firmware on the host and then perform fuzzing. Firmware can be obtained from official websites, device debugging interfaces, or traffic during automatic device updates. Firmadyne [12] successfully emulated 9486 firmwares of the NED by replacing the kernel. FirmAFL [73] implemented full-system emulation-based firmware fuzzing based on Firmadyne. Moreover, P2IM [20], HALucinator [14], μAFL [37] and Fuzzware [61] established a more complete peripheral interaction model to enhance the success rate and fidelity of firmware emulation. However, the manual effort and hardware requirements associated with full-system emulation limit the applicability of this approach. Therefore, partial emulation [51, 71] offers a method for forwarding peripheral API requests to the actual device to complete firmware fuzzing tests. Furthermore, in a well-performing firmware emulation environment, we can migrate applications of advanced gray-box protocol fuzzers [5, 36, 44, 52] by replacing source code instrumentation with binary instrumentation. While emulation-based fuzzing offers code test coverage information and convenient anomaly monitoring and judgment, it requires significant manual effort and can sometimes be infeasible due to missing software or hardware dependencies required by the firmware that are absent in the emulation environment.

For device-based fuzzing, analysts directly connect to the communication interface of the SUT and send test cases to complete fuzzing. Generative fuzzers such as Peach [22], Boofuzz [6], and KittyFuzzer [23] allow users to define templates according to protocol specifications, which include message fields and corresponding mutation methods for generating test cases. This approach is suitable for public protocols with protocol specifications. For proprietary protocols, IoTFuzzer [13] and DIANE [60] proposed to analyze the communication software corresponding to the NED to generate test cases. When the communication software is unanalyzable, analysts can test the NED by using normal communication traffic as seeds and mutating them to generate test cases [24, 53]. Furthermore, to improve the problem of low test efficiency due to the lack of guidance information for direct testing on NEDs, SNIPUZZ [21] and PCFuzzer [43] proposed a method of evaluating response messages to guide seed screening. CHATAFL [48] proposed using the large language model to analyze protocol specifications to help seed generation and mutation. Although device-based fuzzing is more convenient than emulation-based fuzzing, device-based fuzzing requires the purchase of real devices for analysis. Moreover, the program execution information obtained by the device-based fuzzing is scarce, which results in lower test efficiency compared to emulation-based fuzzing.

8.3 Anti-analysis Techniques

As we mentioned in §7, vendors commonly employ reverse engineering mitigation techniques and vulnerability exploit mitigation techniques to enhance the security of their devices. In addition, some anti-analysis techniques have been proposed to prevent the analysis of programs. Symbolic execution, which is frequently used in vulnerability analysis [15, 66], is a program analysis technique that uses abstract symbols instead of precise values as program input variables to obtain the abstract output of each path. Anti-symbolic-execution techniques [62, 65, 70] design encryption or linear obfuscation operate on conditional branches to make the collected constraint information unsolved. Taint analysis is another program analysis technique that monitors the flow of taint

information within a program to detect whether external input can influence key program operations. Anti-taint-analysis techniques [11] disrupt the tracking of tainted data by adding indirection call, implicit flow and time sensitivity.

9 CONCLUSION

In this paper, we present a novel adversarial fuzzing technique, named Armor, that aims to protect NEDs from protocol fuzzing attacks. The proposed approach employs three adversarial primitives, namely delay, fake coverage, and forged exception, to disrupt the fundamental mechanisms of fuzzing, including high throughput, coverage guidance, and efficient detection of exceptions. Based on our observation that normal user inputs adhere to the protocol specification and that there exist cold paths in the program that are seldom executed by normal inputs, we design static and dynamic strategies to decide the insertion location and initialization of adversarial primitives. Our proposed technique effectively resists protocol fuzzing without affecting normal usage, as demonstrated by our extensive evaluations. We show that Armor incurs negligible time overhead and reduces the line coverage for fuzzing by 22%-61% and the number of vulnerabilities found by 57%-89% when compared to the original unprotected program. Furthermore, our approach outperforms the state-of-the-art anti-fuzzing techniques, such as ANTIFuzz and FUZZIFICATION, which can introduce up to 600 times the time overhead and compromise device availability.

ACKNOWLEDGMENT

We thank the associated editor and anonymous reviewers of TOSEM for their valuable feedback. Their insightful suggestions helped us improve this manuscript. This work was supported by the the National Key R&D Program of Ministry of Science and Technology (No. 2021YFB3101803) and the Natural Science Foundation of Beijing (No.V1640354653903).

REFERENCES

[1] AFL. 2020. https://github.com/google/AFL. Accessed 2022-10-20.
[2] AFL++. 2022. https://github.com/AFLplusplus/AFLplusplus. Accessed 2022-10-20.
[3] Ross Anderson and Tyler Moore. 2007. Information security economics–and beyond. In *Annual international cryptology conference*. Springer, 68–91.
[4] Theodoros Apostolopoulos, Vasilios Katos, Kim-Kwang Raymond Choo, and Constantinos Patsakis. 2021. Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. *Future Generation Computer Systems* 116 (2021), 393–405.
[5] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. 2022. Stateful greybox fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3255–3272.
[6] BooFuzz. 2022. https://github.com/jtpereyda/boofuzz. Accessed 2022-10-20.
[7] David J. Brantley. 2005. https://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD303.html. Accessed 2022-10-20.
[8] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 463–469.
[9] Muhammad Arif Butt, Zarafshan Ajmal, Zafar Iqbal Khan, Muhammad Idrees, and Yasir Javed. 2022. An In-Depth Survey of Bypassing Buffer Overflow Mitigation Techniques. *Applied Sciences* 12, 13 (2022), 6702.
[10] G. Casteur, A. Aubaret, B. Blondeau, V. Clouet, A. Quemat, V. Pical, and R. Zitouni. 2020. Fuzzing attacks for vulnerability discovery within MQTT protocol. In *2020 International Wireless Communications and Mobile Computing (IWCMC)*. 420–425. https://doi.org/10.1109/IWCMC48107.2020.9148320
[11] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. 2007. Anti-taint-analysis: Practical evasion techniques against information flow based malware defense. *Secure Systems Lab at Stony Brook University, Tech. Rep* (2007), 1–18.
[12] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. 2016. Towards automated dynamic analysis for linux-based embedded firmware.. In *NDSS*, Vol. 1. 1–1.
[13] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.

[14] Abraham Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. Halucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium*.
[15] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution.. In *USENIX Security Symposium*. 463–478.
[16] Dan Dinculeană and Xiaochun Cheng. 2019. Vulnerabilities and limitations of MQTT protocol used between IoT devices. *Applied Sciences* 9, 5 (2019), 848.
[17] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.
[18] Emil Edholm and David Goransson. 2016. Escaping the fuzz-evaluating fuzzing techniques and fooling them with anti-fuzzing. (2016).
[19] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.
[20] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 1237–1254.
[21] Xiaotao Feng, Ruoxi Sun, Xiaogang Zhu, Minhui Xue, Sheng Wen, Dongxi Liu, Surya Nepal, and Yang Xiang. 2021. Snipuzz: Black-box fuzzing of iot firmware via message snippet inference. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 337–350.
[22] Peach Fuzzer. 2017. https://github.com/MozillaSecurity/peach. Accessed 2022-10-20.
[23] Google. 2019. https://github.com/cisco-sas/kitty. Accessed 2022-10-20.
[24] Serge Gorbunov and Arnold Rosenbloom. 2010. Autofuzz: Automated network protocol fuzzing framework. *Ijcsns* 10, 8 (2010), 239.
[25] Emre Guler, Cornelius Aschermann, et al. 2019. AntiFuzz: Impeding Fuzzing Audits of Binary Executables. In *28th USENIX Security Symposium*. 1931–1947.
[26] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. 2020. SkillExplorer: Understanding the Behavior of Skills in Large Scale. In *29th USENIX Security Symposium (USENIX Security 20)*. 2649–2666.
[27] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. 2022. {RapidPatch}: Firmware Hotpatching for {Real-Time} Embedded Devices. In *31st USENIX Security Symposium (USENIX Security 22)*. 2225–2242.
[28] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 191–205.
[29] Jinho Jung et al. 2019. Fuzzification:Anti-Fuzzing Techniques. In *28th USENIX Security Symposium*. 1913–1930.
[30] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. 2014. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 329–340.
[31] Mark Kirschenbaum. 2020. A PRACTICAL GUIDE FOR CRACKING AES-128 ENCRYPTED FIRMWARE UPDATES. https://gethypoxic.com/blogs/technical/a-practical-guide-for-cracking-aes-128-encrypted-firmware-updates. Accessed 2023-9-20.
[32] George Klees, Andrew Ruef, et al. 2018. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.
[33] Nir Kshetri. 2006. The simple economics of cybercrimes. *IEEE Security & Privacy* 4, 1 (2006), 33–39.
[34] Yingxu Lai, Huijuan Gao, and Jing Liu. 2020. Vulnerability Mining Method for the Modbus TCP Using an Anti-Sample Fuzzer. *Sensors* 20, 7 (2020). https://doi.org/10.3390/s20072040
[35] Ralph Langner. 2011. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy* 9, 3 (2011), 49–51.
[36] Junqiang Li, Senyi Li, Gang Sun, Ting Chen, and Hongfang Yu. 2022. SNPSFuzzer: A Fast Greybox Fuzzer for Stateful Network Protocols using Snapshots. *IEEE Transactions on Information Forensics and Security* 17 (2022), 2673–2687.
[37] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. 2022. μAFL: non-intrusive feedback-driven fuzzing for microcontroller firmware. In *Proceedings of the 44th International Conference on Software Engineering*. 1–12.
[38] Yuekang Li, Guozhu Meng, et al. 2021. Vall-nut: Principled Anti-Grey box-Fuzzing. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering*. IEEE, 288–299.
[39] Lib60870. 2022. https://github.com/mz-automation/lib60870. Accessed 2022-10-20.
[40] libiec61850. 2023. https://github.com/mz-automation/libiec61850. Accessed 2023-10-1.
[41] Libmodbus. 2022. https://github.com/stephane/libmodbus. Accessed 2022-10-20.
[42] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. 2014. *The art of memory forensics: detecting malware and threats in windows, linux, and Mac memory*. John Wiley & Sons.
[43] Puzhuo Liu et al. 2022. Fuzzing proprietary protocols of programmable controllers to find vulnerabilities that affect physical control. *Journal of Systems Architecture* 127 (2022), 102483.
[44] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. 2023. BLEEM: packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4481–4498.

[45] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function code aware fuzz testing of ics protocol. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–22.
[46] Matias Madou, Ludo Van Put, and Koen De Bosschere. 2006. Loco: An interactive code (de) obfuscation tool. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*. 140–144.
[47] Valentin JM Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.
[48] Ruijie Meng, Martin Mirchev, Marcel Böhme, and Abhik Roychoudhury. 2024. Large language model guided protocol fuzzing. In *Proceedings of the 31st Annual Network and Distributed System Security Symposium (NDSS)*.
[49] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. 2021. DICE: Automatic emulation of dma input channels for dynamic firmware analysis. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1938–1954.
[50] Eclipse Mosquitto. 2022. https://github.com/eclipse/mosquitto. Accessed 2022-10-20.
[51] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar 2: A multi-target orchestration platform. In *Proc. Workshop Binary Anal. Res.(Colocated NDSS Symp.)*, Vol. 18. 1–11.
[52] Roberto Natella. 2022. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering* 27, 7 (2022), 191.
[53] Matthias Niedermaier, Florian Fischer, and Alexander von Bodisco. 2017. PropFuzz—An IT-security fuzzing framework for proprietary ICS protocols. In *2017 International conference on applied electronics (AE)*. IEEE, 1–4.
[54] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. HERA: Hotpatching of Embedded Real-time Applications.. In *NDSS*.
[55] 70 Percent of IoT Devices Vulnerable to Cyberattacks. 2014. https://www.securityweek.com/70-iot-devices-vulnerable-cyberattacks-hp. Accessed 2022-10-20.
[56] Top Cyber Attacks on IoT Devices in 2021. 2021. https://firedome.io/blog/top-cyber-attacks-on-iot-devices-in-2021. Accessed 2022-10-20.
[57] Van-Thuan Pham et al. 2020. AFLNET: A Greybox Fuzzer for Network Protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 460–465. https://doi.org/10.1109/ICST46399.2020.00062
[58] Preeny. 2021. https://github.com/zardus/preeny. Accessed 2022-10-20.
[59] Panagiotis Radoglou-Grammatikis, Panagiotis Sarigiannidis, Ioannis Giannoulakis, Emmanouil Kafetzakis, and Emmanouil Panaousis. 2019. Attacking IEC-60870-5-104 SCADA Systems. In *2019 IEEE World Congress on Services (SERVICES)*, Vol. 2642-939X. 41–46. https://doi.org/10.1109/SERVICES.2019.00022
[60] Nilo Redini, Andrea Continella, Dipanjan Das, Giulio De Pasquale, Noah Spahn, Aravind Machiry, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2021. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 484–500.
[61] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. 2022. Fuzzware: Using Precise {MMIO} Modeling for Effective Firmware Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 1239–1256.
[62] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. 2016. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys (CSUR)* 49, 1 (2016), 1–37.
[63] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels.. In *USENIX Security Symposium*. 167–182.
[64] Net Security. 2022. https://www.helpnetsecurity.com/2022/08/29/vulnerability-disclosures-iot-devices/. Accessed 2022-10-20.
[65] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. 2008. Impeding Malware Analysis Using Conditional Code Obfuscation.. In *NDSS*.
[66] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.. In *NDSS*, Vol. 1. 1–1.
[67] David A Solomon, Mark E Russinovich, and Alex Ionescu. 2009. *Windows internals*. Microsoft Press.
[68] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.
[69] Aurélien Vasselle, Philippe Maurine, and Maxime Cozzi. 2019. Breaking mobile firmware encryption through near-field side-channel analysis. In *Proceedings of the 3rd ACM Workshop on Attacks and Solutions in Hardware Security Workshop*. 23–32.
[70] Zhi Wang, Jiang Ming, Chunfu Jia, and Debin Gao. 2011. Linear obfuscation to combat symbolic execution. In *Computer Security–ESORICS 2011: 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12-14, 2011. Proceedings 16*. Springer, 210–226.
[71] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. 2014. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares.. In *NDSS*, Vol. 23. 1–16.
[72] Yingpei Zeng, Mingmin Lin, Shanqing Guo, Yanzhao Shen, Tingting Cui, Ting Wu, Qiuhua Zheng, and Qiuhua Wang. 2020. Multifuzz: A coverage-based multiparty-protocol fuzzer for iot publish/subscribe protocols. *Sensors* 20, 18 (2020), 5194.
[73] Yaowen Zheng, Davanian, et al. 2019. FIRM-AFL:High-Throughput Greybox Fuzzing of IoT Firmware via Augmented Process Emulation. In *28th USENIX Security Symposium*. 1099–1114.

[74] Feilong Zuo, Zhengxiong Luo, Junze Yu, Ting Chen, Zichen Xu, Aiguo Cui, and Yu Jiang. 2022. Vulnerability detection of ICS protocols via cross-state fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 11 (2022), 4457–4468.