# 논문 제목: MQTT security: A Novel Fuzzing Approach

## 요약

사물 인터넷은 점점 더 우리의 삶에 존재하는 개념이다. 지능형(intelligent) 장치의 출현으로 기술이 환경과 상호 작용하여 사회를 더 스마트한 세상으로 이끄는 패러다임 변화가 발생했습니다. 결과적으로 새로운 향상된 텔레메트리 접근법은 모든 종류의 장치를 각가의 사람들과 함께 또는 회사, 인터넷과 같은 다른 네트워크와 연결하는 것처럼 나타난다. 중요한 장치가 필수 서비스를 제공하기 위해 통신 네트워크에 의존하는 점점 더 상호 연결된 세계로 가는길에 이러한 프로토콜과 응용 프로그램의 보안과 안정성을 보장해야 할 필요가 있다. 이 논문에서는 MQTT (Message Queue Telemetry Transport)에 대한 보안 기반 접근 방식에 대해 논의한다. MQTT (Message Queue Telemetry Transport)는 전 세계의 IoT (Internet of Things) 장치를 위한 매우 가볍고 널리 사용되는 메시징 및 정보 교환 프로토콜입니다. 이를 위해 우리는 MQTT 프로토콜에 대해 새롭고 템플릿 (Template) 기반의 퍼징 (fuzzing) 기술을 수행 할 수있는 프레임 워크의 생성을 제안한다. 첫 번째 실험 결과는 여기 제시된 퍼징 기법의 성능이 스마트 시티와 같은 낮은 처리 능력 센서를 가진 네트워크 아키텍처에서 사용하기에 좋은 후보임을 보여주었고, 또한 MQTT를 구현하는 널리 사용되는 응용 프로그램에서이 fuzzer를 사용하면 지금까지보고되지 않은 몇 가지 새로운 보안 결함이 발견되어 보안 취약성을 찾는 도구로서의 유용성을 입증하였다.

## 1. Introduction

오늘날 도시는 지속 가능한 도시 개발, 오염 및 에너지 소비 감소, 안전 등 복잡한 과제에 직면해 있다. IoT(Internet of Things)는 "모든 것이 인터넷에 연결될 수 있다."는 개념을 기반으로 하기 때문에 스마트 시티를 구축하기 위한 핵심 기술로 간주된다. 저렴한 센서 및 기타 장치의 개발과 클라우드 서비스의 채택은 도시의 삶의 질을 향상시키는 새로운 서비스를 개발할 수 있는 새로운 기회를 제공한다. 도시가 성장함에 따라 새로운 IoT 기술을 탐구하는데 관심이 증가하고 있다. IoT 기술이 스마트 도시 건설을 지원하는 방법에 대한 몇가지 예는 다음과 같다.

(1) 자동차의 움직임을 감시하고 에너지를 절약하기 위해 스위치를 켜거나 끌 때 제어할 데이터를 전송하는 센서가 있는 스마트 가로등
(2) 공원의 물 소비량 줄이기
(3) 비상사태 시 시민들의 건강 유지를 위한 실시간으로 의료 기록에 액세스할 수 있는 기능
(4) 실시간으로 데이터를 전송하는 스마트 미터 및 센서에 의해 제어되는 전기의 소비 및 규제

서로 다른 위치에서 통신하고 대량의 데이터를 생성하는 IoT 기술로 인해 정보는 보호하기가 더 여려워지고 있다. 이러한 데이터의 손상된 가용성, 무결성 또는 기밀성은 사람들의 삶에 불리한 직접적인 영향을 미칠 수 있다. 따라서 IoT 장비의 보안, 정보 교환에 사용하는 네트워크 프로토콜 및 해당 장치 용으로 개발된 응용 프로그램을 확인하는 메커니즘을 구현해야한다. 이 논문에서는 IoT 장비간에 교환되는 데이터를 공유하기 위해 널리 사용되는 프로토콜인 MQTT(Message Queue Telemetry Transport) 프로토콜을 구현하는 응용 프로그램의 보안을 향상시키는 프레임 워크를 제안한다. MQTT는 배포가 쉽도록 설계된 publish/subscribe 아키텍처를 사용하는 매우 간단하고 가벼운 메시징 프로토콜이며 단일 서버로 수천 개의 클라이언트를 지원할 수 있다. 또한, MQTT는 불리한 조건에서 신뢰성과 효율성을 제공한다. 이러한 모든 기능을 통해 이 프로토콜은 스마트 장치간에 통신에 가장 많이 사용되는 프로토콜 중 하나이며 이를 기반으로 하는 많은 수의 응용 프로그램은 시간이 지남에 따라 빠르게 증가하고 있다.

MQTT를 구현하는 응용 프로그램의 보안을 테스트하기 위해 fuzzing 이라는 검증 기술을 기반으로 프레임 워크를 만들었다. 퍼징은 예상되지 않은 입력 데이터를 대상 시스템에 보내고 결과를 모니터링함으로써 소프트웨어 응용 프로그램의 취약점을 찾는 테스트 기술이다. 일반적으로 자동 또는 반자동 프로세스로 구성되며, 여기에는 연구중인 시스템으로 데이터를 전송하고 반복적으로 조작하는 작업이 포함된다. 모든 fuzzer는 크게 두 가지 범주로 나눌 수 있다. 하나는 Mutation 기반과 다른 하나는 generation 기반 fuzzer이다. Mutation 기반 fuzzer는 테스트 공간을 생성하기 위해 지곤 데이터 샘플에 변형을 적용하는 반면, generation 기반 fuzzer는 대상 프로토콜이나 파일 형식을 모델링하여 처음부터 테스트 케이스를 만든다. generation 기반 접근법이 보다 복잡하고 시간이 많이 소요됨에 따라 이 논문에서는 template 기반의 novel fuzzing 기법과 함께 mutation 기반의 fuzzing 접근법에 중점을 둔다. 이 기술의 목표는 MQTT 프로토콜을 구현하는 응용 프로그램의 보안 검증을 수행할 때 사용자의 노력을 줄이고 생산성을 높이는 것이다. 이 새로운 기술을 통해 각 네트워크 패킷에 대해 테스트할 필드가 있는 template을 완전히 자동으로 생성할 수 있다. 또는 트래픽을 필터링할 필드 또는 기본적으로 삽입하거나 보내려는 값과 같은 다른 사양을 정의할 수 있다.

논문의 나머지 부분은 다음과 같이 구성된다. 2절에서는 제시된 연구를 이해하는데 필요한 개념을 간략하게 설명한다. 3장에서는 MQTT 프로토콜 보안과 현재 fuzzing 접근법에 관한 관련 연구에 대해 논의한다. 4절에서는 프레임 워크의 기본 요소와 위에서 설명한 개념과 방법의 구현에 대해 다룬다. 5장에서는 fuzzer의 아키텍처를 소개하고, 6장에서는 실험 단계를 결과를 보여준다. 마지막으로 결론을 제시한다.

# 2. Background and Motivation

2.1 Message Queue Telemetry Transport

MQTT는 publisher라고 하는 정보 제공자와 subscriber라고 하는 정보의 소비자간에 느슨한 결합을 가능하게 하는 publish/subscribe 메시징 패턴을 사용한다. 이는 publisher와 subscriber간에 메시지 브로커를 도입하여 이루어진다.

전통적인 point-to-point 패턴과 비교할 때 이 모델의 장점은 publishing 장치 또는 응용 프로그램이 subscribing 장치에 대해 전혀 알 필요가 없으며, 그 반대의 경우도 마찬가지이다. 우리는 이 논문의 개발 과정에서 남아있을 MQTT 필수 개념 세가지를 구별할 수 있다.

(1) Topics

publisher는 topic별로 보낸 메시지를 카탈로그화 해야한다. topic은 메시지를 분류할 수 있는 메시지의 내용 또는 범주를 정의한다. point-to-point 프로토콜에서 특정 주소로 publish/subscirbe 패턴으로 메시지가 전송되는 동안 메시지는 subscriber가 선택한 topic을 기반으로 분산되기 때문에 topic은 중요하다. 특정 topic에 가입하면 해당 topic과 함께 publisher가 보낸 모든 메시지가 subscriber에게 전송된다.

(2) Client

MQTT 클라이언트는 메시지 교환을 위해 브로커에 연결한다. 이들은 topic을 subscribe 해야하며 topic을 제공하여 동일한 브로커에 연결된 다른 엔티티에 정보를 publish 할 수 있다.

(3) Broker

MQTT 브로커는 메시지의 중개자 역할을 하는 서버이다. MQTT 프로토콜 메시지의 형식은 표 1에 표시된 고정 헤더, 가변 헤더 및 페이로드 세 부분으로 구성된다. Fuzzer들은 fuzzing 과정을 수행하기 위해 데이터를 삽입하기 위한 헤더의 필드와 위치를 고려한다.

MQTT는 표 2에 나타난대로 전세계적으로 가장 많이 사용되는 프로토콜 중 하나이다.

## 2.2 Fuzzing Processes

퍼징 프로세스의 단계는 매우 가변적이며 테스트되는 응용 프로그램이나 프로그래머의 경험과 같은 많은 요소에 따라 달라진다. 그러나 분석중인 접근법이나 응용 프로그램에 관계없이 항상 따라야 할 일련의 기본 단계가 있다. 아래에 제시될 도구의 개발에서 다음 단계가 고려되었다.

(1) Identifying an objective

모든 fuzzing 프로세스의 첫 번째 단계는 응용 프로그램, 프로토콜 또는 특정 라이브러리의 기능일 수 있는 대상을 식별하는 것이다. 여기에 있는 목표는 MQTT 프로토콜과 프로토콜을 구현하는 응용 프로그램이다.

(2) Identifying the entry points

거의 모든 악용 가능한 취약점은 사전에 올바르게 검사하지 않고 처리된 사용자 값을 허용하는 응용 프로그램에 의해 발생한다. 입력 벡터를 열거하는 것은 fuzzing 프로세스가 성공하는데 중요한 요소 중 하나이다. 결국 클라이언트에서 대상 시스템으로 보낼 수 있는 모든것을 입력 벡터로 간주해야한다. 여기에는 헤더, 파일 이름, 환경 변수 및 레지스트리 키가 포함된다.

(3) Generating the fuzzing data

일단 입력 벡터가 식별되면, fuzzing 프로세스를 수행하기 위해 적절한 데이터를 생성해야한다. 많은 경우가 생성되어야하기 때문에 테스트 케이스를 생성하는 고도의 자동화가 중요하다.

(4) Executing the test cases

이 단계는 이전 단계와 밀접하게 연결되어 있으며 대상 시스템에서 데이터 패킷을 보내는 프로세스로 구성된다. 이전 단계에서처럼 프로세스 자동화가 필수적이다.

(5) Exception monitoring

fuzzing 프로세스의 중요한 부분은 예외를 모니터링하는 것이다. 수많은 데이터 패킷을 보낸 후 대상 시스템이 손상되는 것은 오류의 원인이 된 특정 패킷을 판별할 수 없는 경우 이점이 없다. 모니터링은 다양한 형식을 취할 수 있으며 대상 시스템과 수행되는 fuzzing 유형과 밀접하게 연결된다.

Table 1: MQTT fixed header.

| Bit-> | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | | Msg type | | | DUP | QoS | | Retain |
| Byte 2 | | | | | Remaining length | | | |

Table 2: Some solutions that use MQTT.

| Brokers | Clients | Smart home |
|---|---|---|
| Mosquitto | CocoaMQTT | Homegear |
| ActiveMQ | emqttc | Domoticz |
| hbmqtt | mqtt-client | Lelylan |
| HiveMQ | M2Mqtt | cul2mqtt |
| Moquette | mqtt_cpp | aqara-mqtt |
| Mosca | mqttex | Home.Pi |
| VerneMQ | Paho | Home Assistant |
| hrotti | rumqtt | pimatic |
| SurgeMQ | hbmqtt | FHEM |

# 3. Related work

3.1 Security in MQTT Protocol

MQTT 보안에 대한 연구는 아직 부족하지만 보안 문제에 대한 초기 연구가 일부 발표되었다. 발생하는 거의 모든 보안 문제는 플포토콜이 기본적으로 동작하는 상태와 관련이 있다. MQTT는 처리 능력이 낮은 장치용으로 설계된 간단한 프로토콜이기 때문에 기본적으로 프로토콜은 심각한 보안 문제가 발생한다는 메시지 교환에 필요한 처리를 최소화하려고 한다. 이러한 단점의 대부분은 적절한 프로토콜 구성으로 해결할 수 있다. 다음은 적절한 프로토콜 구성을 통해 해결할 수 있는 가장 일반적인 보안 문제 중 일부이다.

(1) Lack of authentication

MQTT 프로토콜은 기본적으로 보안 인증 메커니즘을 제공하지 않으므로, 일부 참여자의 신원을 통신 또는 권한이 없는 데이터를 스푸핑할 수 있다. 이 문제는 프로토콜 기능을 적절하게 구성하여 쉽게 해결할 수 있다. 인증과 관련하여 프로토콜 자체는 CONNECT 메시지에 사용자 이름 및 암호 필드를 제공하여 클라이언트가 MQTT 브로커에 연결할 때 사용자 이름 및 암호를 보낼 수 있게한다.

(2) Lack of authorization

MQTT 클라이언트는 브로커에 연결한 후 메시지를 공개하거나 topic에 등록할 수 있다. 인증된 각 클라이언트는 적절한 권한 없이도 모든 topic을 publish하고 subscribe할 수 있다.이는 프로토콜 자체가 이를 수행할 어떠한 메커니즘도 제공하지 않아 브로커와 책임이 있기 때문에 심각한 문제가 될 수 있다. 그럼에도 불구하고 브로커 측에서 topic 권한을 구현하여 쉽게 해결할 수 있다.

(3) Lack of confidentiality

MQTT는 TCP를 전송 프로토콜로 사용한다. 즉, 기본적으로 연결은 암호화된 통신을 사용하지 않는다. 이것은 동일한 네트워크에서 청취하는 공격자가 패킷을 탐지 할 수 있음을 의미한다. 이를 피하기 위해 거의 모든 MQTT 브로커는 일반 TCP 대신 TLS를 사용하여 전체 MQTT 통신을 암호화할 수 있다.

(4) Lack of Intergrity

MQTT 시스템에 신뢰할 수 없는 클라이언트가 있거나 미확인 MQTT 클라이언트가 MQTT 브로커 및 topic에 액세스 할 수 있는 경우, 특히 TLS가 사용되지 않을 때 보낸 메시지의 데이터 무결성을 검사해야한다. MQTT는 교환된 패킷에 무결성을 제공하는 세가지 메커니즘(체크섬, MAC 및 디지털 서명)을 지원한다.

다른 접근법들도 몇몇 저자들에 의해 제시되었다. 또한, 일부 연구는 IoT 장차에서 사용되는 IP 기반 프로토

콜의 일반적인 문제를 처리하려고 시도했으며, 그 중 하나가 MQTT이다. 이러한 경우 저자는 TCP/IP 프로토콜 및 IoT 네트워크에 가장 적합한 보안 아키텍처 및 모델을 포괄할 수 있는 보호 계층을 다루는 등 광범위한 스펙트럼의 일부로 이 유형의 장치의 보안에 중점을 둔다. 기밀 유지외에도, 다른 보안 기능이 다루어졌다. 스마트 장치가 인증 작업을 수행하는 데 비대칭 암호화 알고리즘을 사용할 수 있는 충분한 처리 용량을 가지고 있지 않을 때 해시 함수나 OR 연산과 같은 리소스를 거의 소비하지 않는 연산을 기반으로 하는 새로운 인증 방법을 제안한다. MQTT 프로토콜이 구현할 수 있는 선택적 보안 기능을 강제로 준수하는 방법에 중점을 둔 몇가지 흥미로운 방법이 있다. SecKit은 일련의 보안 정책을 사용하도록 강제하는 모델 기반 보안 툴킷이므로 프로토콜이 기본 구현에서 발견되지 ㅇ낳는 일부 보호 조치를 구현한다.

이 프로토콜은 의도적으로 제기하는 보안 제한 사항에 계속 초점을 맞추고 SSL/TLS와 같은 추가 계층을 추가하여 연결에 관련된 당사자간에 정보를 안전하게 전송하기 위한 프레임워크를 제안하는 연구도 있다. 이 절의 시작 부분에 있는 고려 사항은 프로토콜 보안 결함이 해당 작업, 특히 정보를 교환하는 방식과 관련되어 있음을 보여준다. 이 논문은 MQTT 프로토콜을 구현하는 장치의 검증과 관련하여 MQTT 프로토콜의 보안 향상에 기여하고자 한다. MQTT 프로토콜은 구현하는 응용 프로그램이 패키지를 잘못 처리하면 위의 보안 조치가 충족될 때에도 서비스 거부 또는 임의 코드의 원격 실행과 같은 심각한 보안 오류가 발생할 수 있다. 올바르지 않거나 예기치 않은 데이터를 수신했을 때의 동작을 확인하기 위해 연결의 다른 부분(클라이언트 및 브로커)에서 이러한 프로토콜을 구현하는 응용 프로그램을 평가하면 심각한 보안 침해를 피할 수 있다.

### 3.2 Modern Fuzzing

이전 섹션에서도 언급했듯이 , IoT Systems 책에서 Modbus 프로토콜을 위해 제시된 것과 같은 IoT fuzzing을 포함하여 다양한 유형의 fuzzing 및 이 기술을 수행하는 많은 방법이 있다. 그러나 일반적으로 Aitel이 논문에서 말했듯이, 현대 fuzzing은 전통적인 fuzzing과 관련된 세가지 주요한 문제를 해결하려고 시도한다.

(1)네트워크 프로토콜이 클라이언트와 서버가 구현된 API에 의해 정의도니는 경우, 미리 정의된 이러한 함수가 전송된 데이터를 확인하여 결과적으로 fuzzing 프로세스에 간접적인 영향을 줄 가능성이 높다.
(2) 프로토콜에 관한 완전한 정보가 주어지더라도 프로토콜을 위한 클라이언트를 만드는 것은 상당한 사업이 될 수 있으며, 클라이언트는 비슷한 성질을 가진 프로토콜조차도 다른 프로토콜로 이식할 수 없는 경우가 겨의 없다.
(3) 테스터는 종종 공격을 받는 프로토콜이나 프로토콜이 깨지면 프로토콜에 대한 정보는 제한적이다.

이를 통해 달성되는 것은 모든 블록이 닫히면 하위 계층의 제어 필드가 자동으로 다시 계산되므로 결과를 처리할 필요가 없다는 것이다.

MQTT 프로토콜을 fuzzing하는 특정 topic에 초점을 맞추면, 그에 대한 참조나 도구가 거의 없다. 저자가 증명할 수 있는 유일한 공개 도구는 mqtt-fuzz이다. 이 유틸리티의 주요 유틸리티는 복잡성을 너무 높이지 않고 빠르고 전통적인 프로토콜로 프로토콜을 검증하는 것이다. 또한 MQTT 프로토콜에 대한 다른 fuzzing 또는 형식 테스트 방법이 제시되었지만 다른 목적을 가지고 있다. 이는 저자가 유한 상태 머신 및 레이블이 지정된 전환 시스템을 통한 네트워크 프로토콜 검증의 공식적인 방법을 논의하는 경우이다. MQTT의 대부분의 구현이 표준을 충족시키지 못하는 것을 보여주는데 중점을 둔다. GCFuzzing은 ZigBee를 위한 fuzzing 알고리즘으로, 효율적인 테스트 케이스 생성에 중점을 둔다.

### 3.3 Proxy Fuzzing

Proxy fuzzing은 현재의 한꼐로 인해 광범위하게 연구된 기술이다. 이 기술과 관련하여 ZAP Proxy, Burp Proxy, ProxyFuzz와 같은 일부 작업이 수행되었다. 이 모든 작업에서 공통적으로 갖는 것은 fuzzer가 클라이언트와 서버 사이의 연결 중간에 배치되어 릴레이 에이전트 역할을 수행하는 것이다. 이 작업을 효율적으로 수행하려면 클라이언트와 서버를 수동으로 또는 자동으로 구성해야한다(예: ARP 스푸핑).

이를 해결하기 위해 Boofuzz, SNOOZE 및 KiF와 같은 현대의 fuzzing 도구는 프로토콜을 길이 필드와 데이터

필드로 구분하고 사용자에게 이러한 도구를 만들지 않고 프레임 워크를 제공하는 블록 기반 접근 방식을 제안한다. 하위 레이어의 제어 필드(길이 또는 체크섬 등)에 대해 걱정할 필요가 없다. 이것은 현재의 대부분의 프레임워크가 사용하는 접근법이며 매우 간단한 기초를 가지고 있다. 상위 계층이 응용 프로그래미고 하위 계층이 실제 계층인 여러 계층으로 구성된 네트워크 패킷을 사용할 수 있는 경우 응용 프로그램 계층 중 하나에서 fuzzing 테스트를 수행하려고 한다. 테스트 값을 입력하고 메시지를 전송하면 삽입된 값이 처리되기 전에 서버에 도달할 때 올바르게 옵데이트되지 않으면 패킷 거부를 초래하는 제어 필드가 기본 계층에 포함될 수 있다. 이 문제를 해결하기 위해 블록이라고 하는 일부 구조가 제안되었다. 그것들에서 일련의 변수는 특정 크기를 차지하는 프레임 워크에 의해 이전에 정의되어진 그룹화가 된다. 이 변수들의 집합은 블록을 형성하고 블록들은 다음과 같이 열리고 닫힐 수 있다.

서버는 프록시의 주소에서 서로를 찾는다. 그래서 클라이언트는 프록시를 서버로보고 그 반대의 경우도 마찬가지이다. 이 fuzzing 방법은 사용의 단순성과 같은 이전 프로세스보다 몇 가지 개선 사항을 제공한다. 그러나 이 기법을 구현하기 위한 도구가 거의 알려지지 않고 매우 기본적인 fuzzing 기술을 도입하는 이유는 구현이 어렵기 때문이다. 이 기술을 특허를 이끌어 냈다.

```
s_block_size_binary_bigendian_word("somepacketdata");
s_block_start("somepacketdata");
s_binary("01020304");
s_block_end("somepacketdata");
```

# 4. Methods

이 센션에서는 MQTT 프로토콜을 위한 fuzzing 도구를 만드는 새로운 접근법과 함께 이전 섹션에서 언급한 방법을 구현한 방법을 보여줄 것이다.

### 4.1 Fuzzing MQTT Messages

프로토콜을 구현하는 응용 프로그램이나 응용 프로그램을 fuzzing하는 프로세스는 공용 문서 또는 리버싱 기술을 통해 상기 프로토콜의 사양을 어떤식으로든 알고 있어야 한다. 패키지의 바이트를 해석하고 패키지의 바이트를 해석할 수 있게되면 정보를 삽입하는데 필요한 패키지와 필드를 선택하여 해당 패키지와 프로세스를 처리하는 응용 프로그램이 올바르게 작동하는지 확인해야한다. MQTT의 경우 스펙이 공개되어 있으므로 리버싱 프로세스가 필요하지 않는다. 따라서, 교환되는 패키지 유형과 가변 헤더 및 페이로드(표 3)에 있는 필드에 대해서만 특정 문서를 조사하면된다.

프로토콜에 의해 교환되는 패킷 유형에 대해 좀 더 자세히 살펴보면 PUBLISH 메시지는 대부분의 전송되는 것으로 보이므로 대부분의 정보가 전송되므로 가장 많은 처리가 프로토콜을 구현하는 응용 프로그램. 이러한 유형의 패키지(PUBLISH, CONNECT, SUBSCRIBE 등)를 식별한 후에 가변 헤더(표 4)를 연구하여 필드의 유형과 필드의 위치를 바이트로 선택하며, 테스트 케이스를 삽입하여 fuzzing 프로세스를 수행한다. 테스트 케이스가 삽입된 필드를 선택한 후, 테스트 케이스를 삽입하면 다시 계산되는 제어 필드를 찾고, 마지막으로 "즉각적으로" 필터링할 수 있도록 패키지를 명확하게 식별하는 필드를 찾는다.

### 4.2 Advance Proxy Fuzzing

fuzzing 프로세스를 적용하기 위해 이전 섹션에서 설명한 fuzzing proxy 기술을 사용한다.

우리가 이미 말했듯이, 이 기술을 널리 퍼진것이 아니며 그것을 수행하는 도구는 시대에 뒤떨어져 있으며 현대 기술에 비해 많은 결함을 가지고 있다. 그러나 이 기술을 적용할 때의 장점을 깊게 연구하면 현대적인 fuzzing에 의해 제시된 몇 가지 결함을 해결할 수 있음을 검증할 수 있다. 이러한 결함은 다음과 같다.

#### 4.2.1 Fuzzing Different Components of the connection

일반적으로 현재의 fuzzing 도구는 연결의 일부 지점을 확인하기 위해 설계되었다. 즉, 도구를 사용하여 특정 서버를 테스트하면 일반적으로 클라이언트에서 프로세스를 반복하는데 사용할 수 없으며 최소한 프레임 워크의 구조를 수정하는데 많은 노력을 기울이지 않아도 된다. 이 문제점에 대한 솔루션은 MQTT

프로토콜을 구헌하는 어플리케이션의 보안 검증을 위해 사용자 측의 노력을 줄이는 것이 목적이기 때문

에 제시되는 툴의 주요 특징 중 하나를 표시한다. Proxy 기술을 사용하면 fuzzer가 통신의 중간에 있기 때문에 주요 목적은 다른 구성 요소간에 순환하는 패키지이다. 따라서 fuzzing 도구는 특정 서버 또는 클라이언트 용으로 작성된 것이 아니라 주어진 패키지 세트 용으로 작성되었다. 패키지 사양은 프로토콜을 구현하는 모든 응용 프로그램의 표준이므로 퍼징 프로세스는 테스트를 수행하는 연결 지점(예: 클라이언트, 브로커 등)에서 완전히 교환된다.

#### 4.2.2 Fuzzing Messages Based on Previoous Responses

일부 상황에서는 프로토콜의 특정 패킷에 fuzzing을 적용하여 값이 대상 시스템에서 올바르게 처리되는지 여부를 판별할 수 없다. 일부 필드는 이전 메시지를 기반으로 하기 때문이다. 이전에 서버에서 보낸 임의의 핸들 필드가 있는 패키지의 특정 값을 테스트하려는 경우 연결을 설정하고 이 패키지 유형을 계속 보내면 된다. 잘못된 핸들 필드를 가지며 대상 응용 프로그램은 값을 처리하지 않으므로 어떤 경우에도 fuzzing 프로세스가 수행되지 않는다.

이것은 proxy 접근법으로 해결된 또 다른 문제이다. fuzzer가 필터링하고 처리한 메시지는 합법적인 연결을 설정하는 합법적인 클라이언트 및 브로커에서 가져온 것이기 때문에 한쪽에서 다른 쪽으로 이전에 전송된 필드를 그대로 유지하고 적절한 값을 가진다.

### 4.3 Template-Based Fuzzing

본 논문에서는 2.1절과 2.2절에서 제시된 문제를 해결하기 위한 새로운 Template 기반 fuzzing 기법을 제시한다. 앞서 설명한 것처럼 현재의 fuzzing 도구는 블록 기반 기술을 사용하여 제어 필드를 자동으로 다시 계산하여 사용자 작업을 단순화하는 방법을 사용한다. 그렇게더라도 이 방법은 특정 프로토콜에 대한 보안 검사를 수행하려는 사용자에게 매우 복잡한 것으로 입증되고 있다. 아래에 표시된 코드는 요즘 널리 사용되는 Boofuzz라는 프레임 워크를 통해 매우 간단한 프로토콜(FTP)에서 네가지 메시지에 fuzzing을 적용할 수 있는 작은 프로그램을 구현하는데 필요한 코드를 나타낸다. Sulley는 SPIKE의 영향을 많이 받는다. 알 수 있듯이 이러한 유형의 프레임 워크에서 복잡한 프로토콜의 정의는 도구 자체 및 프로토콜의 전체 사양에 대한 철저한 지식외에도 지루한 작업이다.이 시점에서 Template 기반 접근법이 유용할 것이다.

Template 기반 접근법은 다음과 같이 동작한다.

(1) 이 도구는 마치 proxy 기술을 사용하여 스니퍼인것처럼 통신 도중에 대기한다. 사용자는 이전에 일련의 매개 변수를 제공해야만 매개 변수를 통과하는 패킷이 필터링된다. 이들은 이전 섹션에서 논의된 필드이다.

(2) 사용자는 클라이언트와 그가 모호하게 표현하려는 프로토콜의 합법적인 서버 사이에서 트래픽을 생성한다. 이전 지점에서 지정된 패킷이 도구에 의해 차단되면 필터링되고 처리된다.

(3) 패키지를 처리한 후 다음 형식으로 .json Template이 자동 생성된다.

Template의 이 부분은 MQTT 패키지의 MQTT publish 레이어를 표시한다. 보시다시피 패키지의 각 필드가 나타나고 두개의 추가 속성이 각 필드에 추가된다 (fuzzable 및 recalculate). 사용자가 패키지의 특정 필드에 fuzzing을 적용하려면 fuzzable 속성을 true로 지정하여 수정해야한다. 또한 사용자는 패킷 일관성을 유지하기 위해 자동으로 다시 계산되는 것으로 간주되는 필드의 다시 계산 속성에 값 true를 할당해야한다. 이 도구는 fuzzables로 표시된 필드에 확인 값을 자동으로 입력하고 recalculate 플래그가 true로 설정된 패키지의 모든 필드를 다시 계산한다.

그림 1에서 볼 수 있듯이 템플릿의 생성 시간은 비교적 빠르며 생성 알고리즘은 $O\ (n)$로 생성된 템플릿의 수에 관계없이 생성 시간은 일정하게 유지된다. 따라서, 사용자가 도구 또는 프로토콜 자체에 의해 사용되는 구조의 세부 사항을 알 필요가 없기 때문에, 사용자가 fuzzing을 적용하고자하는 필드 외에도 세번째 문제가 해결된다. Template를 수정하기 위해 사용자는 특별한 도구가 필요하지 않다. 이것은 .json 구조가 유지되는 한 일반적인 텍스트 편집기로 편집할 수 있다.

### 4.4 Test Cases Generation

응용 프로그램이 테스트 될 값을 생성하는 것은 fuzzing 프로세스의 중요한 부분이다. 종종 사용자는 외부 응용 프로그램과 함께 생성된 사용자 지정 dictionary를 사용하여 테스트를 실행하는 데 관심이 있다. 반면에 사용자는 특정 지식을 가진 테스트 케이스의 자동 생성에 관심이 있다. 제안된 도구는 두가지 방법을 모두 구현하고 두 경우 모두 사용의 단순성을 유지하려고 시도한다. 이것이 어떻게 구현되는지에 대한 자세한 내용은 다음 섹션에서 설명한다.

Table 3: Types of MQTT messages.

| Packet | Description |
|---|---|
| CONNECT | Connect to the server |
| CONNACK | Ack of connect msg |
| PUBLISH | Publish a topic |
| PUBACK | Ack of publish msg |
| PUBREC | Publication received |
| PUBREL | Publication sent |
| PUBCOMP | Publication completed |
| SUBSCRIBE | Client subscription |
| SUBACK | Ack of subscribe msg |
| UNSUBSCRIBE | Unsubscribe petition |
| UNSUBACK | Ack of unsubscribe msg |

Table 4: Publish packet variable header nonnormative example.

| Byte position | Description |
|---|---|
| *Topic name* | |
| Byte 1 | Length MSB (0) |
| Byte 2 | Length LSB (3) |
| Byte 3 | "a" (0x61) |
| Byte 4 | "/" (0x2F) |
| Byte 5 | "b" (0x62) |
| *Packet identifier* | |
| Byte 6 | Packet identifier MSB (0) |
| Byte 7 | Packet identifier LSB (10) |

```
def main():
    session = Session(
        target=Target(
        connection=SocketConnection("127.0.0.1",

        8021,proto=\tcp")))
```

```
    s_initialize("user")
    s_string("USER")
    s_delim(" ")
    s_string("anonymous")
    s_static("\r\n")
    s_initialize("pass")
    s_string("PASS")
    s_delim(" ")
    s_string("james")
    s_static("\r\n")
    s_initialize("stor")
    s_string("STOR")
    s_delim(" ")
    s_string("AAAA")
    s_static("\r\n")
    s_initialize("retr")
    s_string("RETR")
    s_delim(" ")
    s_string("AAAA")
    s_static("\r\n")
    session.connect(s_get("user"))
    session.connect(s_get("user"),s_get("pass"))
    session.connect(s_get("pass"),s_get("stor"))
    session.connect(s_get("pass"),s_get("retr"))
    session.fuzz()
```

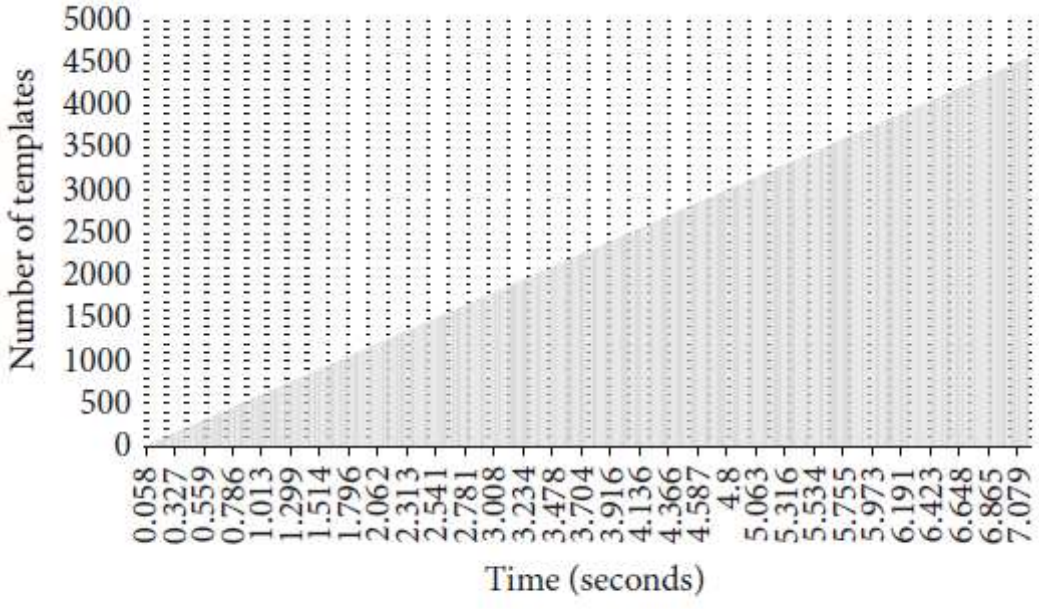


FIGURE 1: Time used by the application to generate templates.

## 5. System Design

이 절에서는 우리의 설계에 동기를 부여하는 설계 고려 사항에 대해 논의한 다음 MQTT fuzzer 시스템 아키텍

처에 대해 설명한다.

5.1 Architecture of the Fuzzer

이 섹션에서는 도구의 디자인과 구현의 관점에서 도구의 구조에 대해 설명한다. 우리는 그것이 구성되어 있는 주요 모듈과 그 기능에 대해 살펴볼 것이다. 또한 fuzzing 프로세스를 만족스럽게 하기 위해 도구가 수행해야 하는 몇 가지 2 차 동작에 대해 설명한다. 그림 2는 일반적인 아키텍처를 보여준다. 이 도구는 다음과 같은 모듈로 구성되어 진다.

- Mitmfuzzer

  Mitmfuzzer 모듈은 나머지 응용 프로그램 기능이 호출되는 드라이버이다. 그안에는 사용자가 입력한 인자가 구문 분석(parsed)된다. 이것은 파이썬 모듈 argparse를 사용하여 수행된다. 또한, 주어진 상황에서 도구의 활동 상태를 보여주는 작은 인터페이스를 제공한다.

- Sniffer

  Sniffer 모듈은 이 도구의 주요 기능 중 하나이다. 패키지를 필터링하고 처리하기 위해 연결 중간에서 위치하여 듣고 있어햐 한다. 따라서 사용자가 지정한 template을 필터링하여 나중에 template을 생성한다. 이 모듈의 구현의 핵심은 많은 수의 프로토콜을 지원하는 네트워크 패킷의 하위 수준 처리를 위한 프레임 워크 인 Scapy를 기반으로 한다. 이 모듈이 사용자가 선택한 패키지를 감지하면 특정 형식으로 직렬화하여 파이썬 프로그래밍 언어를 사용하여 처리할 수 있게 한다. 이 패키지는 template 생성을 위해 template 모듈에 제공한다.

- Template

  이 모듈은 스니퍼 모듈에서 특정 형식의 패키지를 받고 이를 처리하여 .json 형식으로 template를 생성한다. 생성한 template는 template 디렉토리에 저장되며 나중에 fuzzer에서 사용되어 fuzzing 및 recalculate 될 필드와 패키지를 식별한다.

- Fuzzer

  Fuzzer 모듈은 테스트 케이스의 리스팅, 패킷 필터링, 생성 및 삽입 프로세스를 수행하므로 도구의 가장 중요한 모듈이다. 이 모듈의 입력은 template 모듈에 의해 이전에 생성된 template 파일이다. iptables 및 nfqueue 사용을 통해 모듈은 마치 스니퍼인 것처럼 통신을 청취하고, 소개된 template로 식별되지 않은 패키지를 재지정하고 template와 일치하는 패키지를 필터링 및 처리한다. 일치하는 패키지가 처리되면 모든 필드가 template의 필드와 비교되어 사용자 중 하나가 fuzzing되도록 선택되어있는지 여부를 확인한다. 하나 이상의 필드의 경우, 모듈은 사용자가 테스트 프로세스를 수행하기 위해 사용자의 정의 dictionary를 입력했는지 여부를 검사한다. 그렇다면 모듈은 사용자가 제공한 테스트 케이스 중 하나를 검색하여 검증을 위해 표시된 필드에 입력한다. 사용자가 사용자 정의 테스트 케이스를 제공하지 않은 경우, 모듈은 Radamsa를 호출하여 valid case/field name directory에 있는 유효한 테스트 케이스가 있는 파일을 매개 변수로 전달한다. Radamsa는 소프트웨어 검증을 위해 특별히 설계된 stock generator이다. 올바른 데이터를 포함하는 샘플 파일을 읽은 다음 일련의 알고리즘을 통해 이 데이터를 변경하므로 일부 정보가 제공되므로 생성된 결과가 오류로 이어질 가능성이 더 크다. Radamsa는 자동으로 50가지의 테스트 케이스를 생성하며, 모듈을 필드 중 하나를 입력하여 fuzzing 하도록 한다. 모듈에 의해 수행된 테스트 케이스의 생성은 연속적이고 무한이며, 모듈이 생성된 50개의 테스트 케이스를 소모하면 자동으로 Radamsa를 호출하여 또 다른 50개의 새 사례를 생성한다.

- Scapy

  Scapy는 많은 수의 네트워크 프로토콜을 지원하는 패킷 조작을 위한 라이브러리이다. 응용 프로그램의 핵심 요소 중 중요한 부분을 차지하기 때문에 응용 프로그램 아키텍처에서 이름을 지정해야한다고 간주되었다. 광범위한 프로토콜 지원외에도 Scapy의 장점은 블록 기반 접근 방식을 사용한다는 점이다. 즉, 패키지의 필드 중 하나를 수정하면 길이와 기타 제어 필드를 매우 간단하고 자동으로 다시 계산할 수 있다. 패키지가 Fuzzer 모듈에 도착하면 처리를 위해 Scapy로 보내진다. 그런 다음 Scapy는 패키지를 나타내며 조작하기 쉬운 구조를 반환한다. MQTT 패키지 조작을 마친 후, Scapy는 조작된 패키지를 취하는데, 이는 길이 필드(텍스트가 삽입 또는 삭제된 경우) 또는 체크섬 필드(패키지의 바이트가 있는 경우)와 같은 제어 필드의 불일치로 인해 올바르지 않을 수 있다. 블록 기반 접근 방식을 사용하여 몯느 제어 필드를 다시 계산하고 데이터를 원래 패키지에 캡슐화한다. 이 패킷을 Fuzzer 모듈로 전달하고 Fuzzer는 이를 합법적인 응용 프로그램으로 전달한다. Scapy가 MQTT 프로토콜을 지원하지 않았음을 강조할 가치가 있으며, 이 논문에서 연구 대상 프로토콜 객체이므로 MQTT에 지원을 추가함으로써 제공되는 라이브러리를 확장했다. 현재 개발된 모듈은 Scapy의 공식 저장소의 일부이다.

5.2 Test Cases Generation Implementation

이 절에서는 프레임 워크의 구현 단계에서 테스트 사례의 자동 생성이 어떻게 적용되었는지 설명한다.

5.2.1 Automatic Generation of Test Cases

도구 아키텍처의 설명에서 설명한 것처럼 테스트 사례의 자동 생성은 Radamsa라는 외부 응용 프로그램에 의해 수행된다. 이 응용 프로그램은 CVE-20073641 및 CVE-2007-3644 (아카이브 읽기 지원 형식 tar.c 라이브러리 취약점), CVE-2008-6536 (7-zip 프로그램 취약점) 및 CVE-2010-2482 (LibTIFF 3.9.4 취약점)과 같은 취약점을 발견하는 데 사용 된 것으로 알려져 있다. 제안된 도구가 이 모듈을 사용하는 방법은 다음과 같다. 도구의 디렉토리에 두 개의 중요한 폴더가 있습니다. 유효 케이스의 하위 디렉토리 집합과 조사 할 패키지의 각 필드에 대한 디렉토리.이러한 하위 디렉토리에는 해당 필드에 대해 올바른 데이터가있는 하나 이상의 샘플 파일이 있다. 이것들은 Radamsa에게 제공되어 변이를 일으켜 테스트 케이스를 생성한다. 반면에 fuzz-cases라는 또 다른 디렉토리가 있는데,이 디렉토리 안에는 특정 패키지에서 각 필드가 퍼지되도록 디렉토리가 만들어져 있다. Radamsa는 내부의 모든 필드 별 테스트 케이스를 자동으로 생성하므로 도구가 패키지를 검색하여 패키지에 삽입한다.

5.2.2 Using Custom Test Cases

테스트 케이스의 자동 생성을 사용하는 대신 다른 툴이나 수동으로 생성된 케이스 세트를 사용하려는 경우 사용자는 위에 설명된 디렉토리 구조에서 다음 단계를 수행하여 직접 수행할 수 있다.

(1) fuzz-cases 디렉토리에서 모호하게 만든 필드의 정확한 이름을 가진 하위 디렉토리를 만듦

(2) 생성된 하위 디렉토리 안에 파일마다 하나씩 모든 테스트 케이스를 입력(파일에 주어진 순서나 이름을 관련이 없다.)

# 6. Experimentation and Results

이 섹션에서는 오늘날 널리 사용되는 일련의 응용 프로그램에 이 도구를 적용한 결과를 제시한다. 제시된 모든 테스트 시나리오는 통제된 환경에서 수행되었다. 테스트 도구는 오픈 소스이며 무료로 사용할 수 있다.

6.1 Performance Considerations

이 섹션에서는 도구의 성능에 미치는 영향을 고려하였다. 이 절은 제시된 도구의 다양한 기능과 각 기능이 성능에 미치는 영향을 평가하는 몇 가지 하위 섹션으로 구분된다.

6.1.1 Packet Processing

이전 섹션에서 설명한 것처럼 이 도구는 클라이언트와 브로커 간 통신 중간에 위치하며 두 클라이언트 사이에 흐르는 모든 네트워크 패킷을 수정하여 proxy 퍼징 기술을 적용한다. 이 때문에 도구의 많은 처리 부하는 패킷 데이터 필드에 테스트 사례를 삽입하는 등의 처리를 이해하고 이전 계층의 모든 제어 필드를 다시 계산하는 것과 같은 즉석에서 패킷의 수정 및 처리에 해당한다.

이를 고려하여 우리가 측정한 측명 중 하나는 패키지 당 처리 시간이다. 그림 3에서 볼 수 있는 바와 같이, 테스트 케이스가 삽입되는 각각의 패키지의 처리 시간은 비교적 일정하게 유지되며, 테스트 케이스가 삽입됨으로써 약간의 변화가 있다. 테스트 케이스의 길이가 길면 더 많은 필드를 다시 계산해야하므로 처리 시간이 길어진다. 그래프 구성을 위해 1300개의 네트워크 패킷 하위 집합이 고려되었으며 평균 처리 시간은 0.003699 초이다. 이것은 안정적인 연결을 유지하기 위한 수용 가능한 시간으로 간주될 수 있다.

처리 시간이 각 패킷에 대해 계산되면, 처리된 후 100 패킷 세트의 도달 지연이 계산된다. 이것은 패킷의 전체 처리 시간이 테스트 케이스를 삽입하고 하위 계층의 모든 제어 필드를 다시 계산하는 것뿐만 아니라 커널 공간에서 사용자에게 패킷을 전송함으로써 발생하는 지연을 고려해야하기 때문에 수행되었다. 패키지를 사용자 공간에서 커널 공간으로 전송하여 전송할 수 있도록 하고, 네트워크를 통해 전송되는 걸리는 시간을 늘리는 등의 작업을 수행할 수 있다.

그림 4에서 볼 수 있듯이, fuzzing된 패킷의 통과 시간은 합법적인 패킷의 시간에 대해 약 90% 증가하며 총 지연 시간은 0.0013085 초로 과도한 지연없이 연결을 안정적으로 유지할 수 있다.

6.1.2 Fuzzer Load and CPU consumption

이전 섹션에서는 도구가 생성할 수 있는 template의 수와 template를 생성하는데 걸리는 시간을 보여주었다. 또 다른 중요한 성능 척도는 fuzzer가 시간 단위당 삽입할 수 있는 테스트 케이스의

수와 호스트 시스템의 CPU 사용량이다.

fuzzer에는 여러 가지 사용자 정의 기능이 있어 삽입된 테스트 케이스 사이의 시간을 선택하여 모든 것을 가능한 빨리 또는 수시로 삽입할 수 있다. 호스트 시스템의 CPU 소비를 평가하기 위해 삽입된 케이스 간의 다른 기간이 고려되었다.

그림 5에서 볼 수 있듯이, fuzzer를 수용하는 기계의 CPU 소비는 삽입된 각 테스트 케이스 사이에 남겨진 지연 시간에 따라 상당히 다르다. 따라서 리소스가 적은 환경에서 도구를 사용자 정의할 수 있다.

### 6.2 Application Scenarios

"응용 프로그램 시나리오"라는 용어는 연결에서 해당 요소에 fuzzing을 적용하기 위해 도구에서 제공하는 가능성을 나타낸다. 이전 섹션에서 설명한 것처럼 채택된 접근 방식을 사용하면 프로토콜 연결의 여러 지점을 모호하게 할 수 있다. 다음은 MQTT 프로토콜을 기반으로 하는 일련의 사용 사례이다.

(1) Pub-fuzzer-broker-Sub

이 경우, 도구는 메시지를 publish하는 클라이언트와 브로커 사이에 배치되어, 도구는 클라이언트에서 서버로 흐르는 메시지와 서버에서 publish중인 클라이언트로 전송되는 메시지를 모호하게 할 수 있다.

(2) Pub-broker-fuzzer-Sub

이 경우, 시나리오는 조금 변할 것이다. 즉, 브로커와 subscribe된 클라이언트 사이에 메시지를 수신하기 위해 대기하는 fuzzer가 있다. 이 도구는 브로커의 메시지를 청취중인 클라이언트로 보낸다. 그리고 클라이언트에서 브로커로 보내는 메시지는 일반적으로 확인, 응답된다.

### 6.3 Results

현재 브로커 및 클라이언트 중 일부에 이 도구를 적용한 후 fuzzer는 서비스 거부를 유발했으며, 다른 유형의 공격 기술을 수행할 수 있는 잠재적인 악용 가능성이 있는 여러 실패를 감지할 수 있었다. 이러한 실패 중 일부는 다음과 같다.

(1) fuzzing 패키지를 잘못 처리하고 응용 프로그램을 중단시키는 Jaba 예외를 throw 한 후, MOQUETTE broker v0.10에 대한 서비스 거부

(2) 연결 재설정을 시작한 fuzzing 패키지를 구문 분석(parsed)한 후 브로커\ MOQUETTE v0.10에 의해 들어오는 연결을 처리하는 중 오류가 발생

(3) 브로커에서 fuzzing 메시지를 수신할 때 특정 주제에 가입한 MOSQUITTO 클라이언트 v1.4.11의 서비스 거부

이는 전 세계의 IoT 장치에서 널리 사용되는 응용 프로그램에서 실제 사용되는 fuzzing 방법이 실제 결과를 제공하므로 주어진 네트워크의 장치가 최소 보안  표준을 충족하는지 확인하기 위한 보안 수단으로 사용될 수 있음을 보여준다.

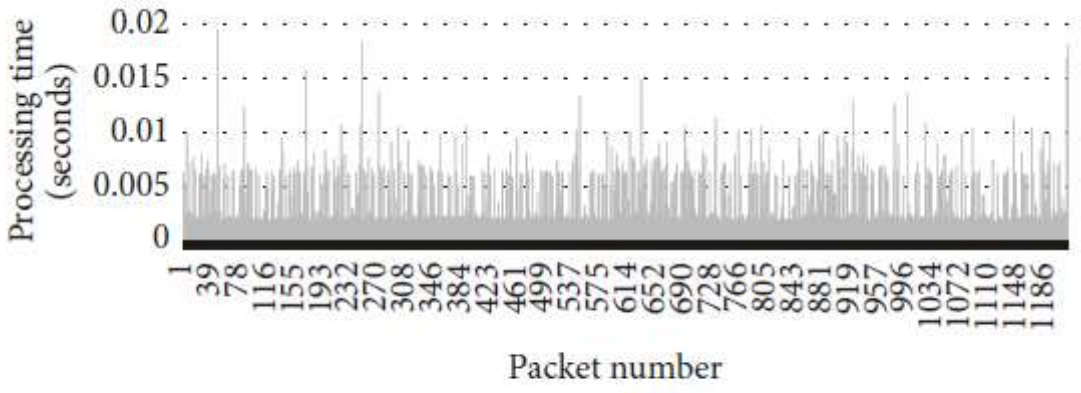


FIGURE 3: Processing time per package.

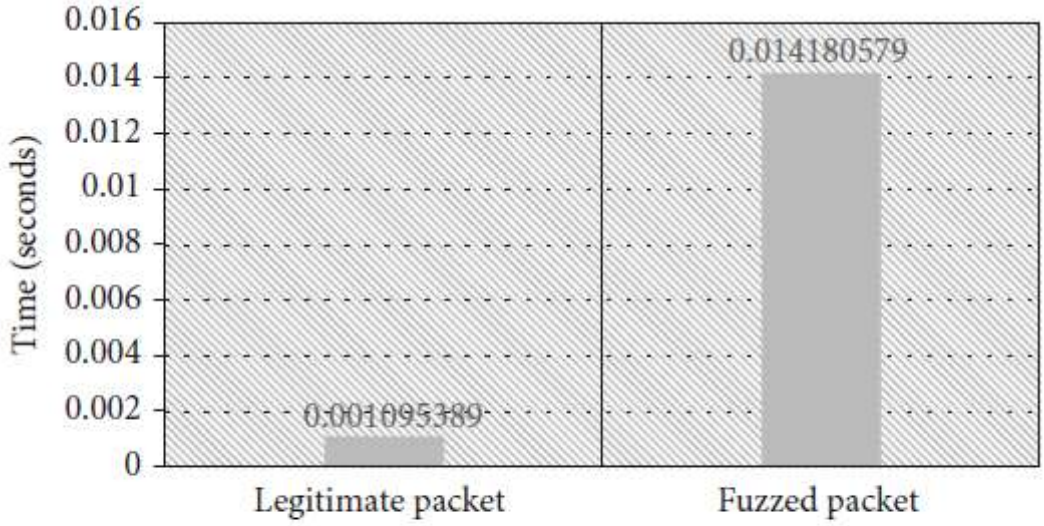


FIGURE 4: Difference between the transit time of a legitimate and a fuzzed package.

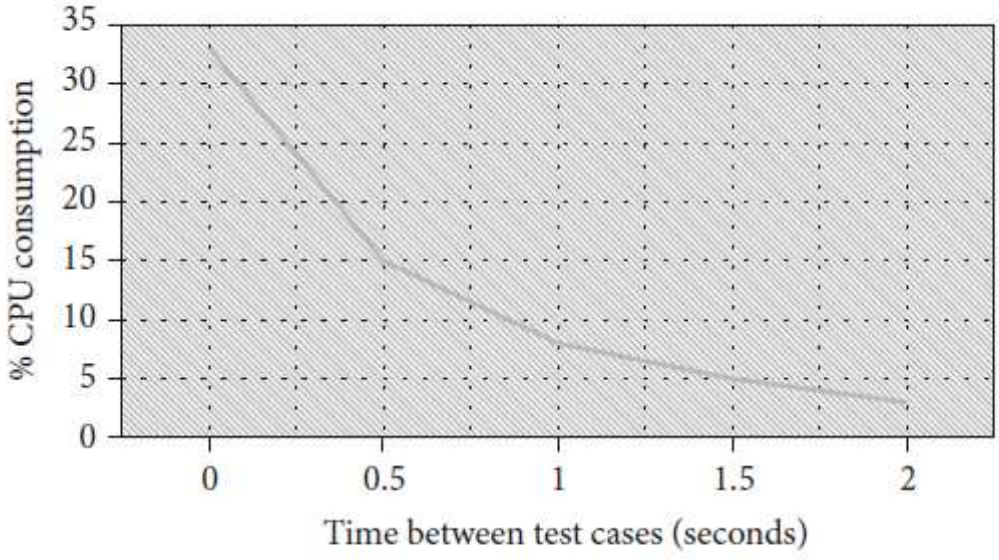


FIGURE 5: CPU consumption in relation to delay time between test cases.

## 7. Conclusion

이 작업의 목적은 IoT 장치의 보안 향상에 기여했으며 특히 정보를 교환하기 위한 통신 프로토콜로 Internet of Things(MQTT)에서 널리 사용되는 프로토콜을 구현하는 응용 프로그램에 기여하였다. 이를 위해 MQTT에 대한 보안 테스트를 위한 프레임 워크를 개발했다. 이 도구는 template에 기반한 새로운 퍼징 기술을 구현한다. 이 기술은 이전에 사용되지 않았던 지식에 기반한다. 여기에 제시된 퍼징 기법은 현재 fuzzing의 결함을 개선하여 기여한다. 이 프레임 워크의 중요한 공헌은 적응 노력을 하지 않고 연결의 다른 지점을 모호하게하는 유연성을 제공한다는 것이다. 이 논문에서 설명된 다른 기여들 중에서도 이전에 제공된 패킷을 기반으로 패키지를 fuzzing 할 수 있으며 template을 교한하여 이식성 및 오류 보고를 용이하게 할 수 있다는 점을 강조할 필요가 있다. 마지막으로, 이 기술은 template 기반 접근법을 사용하여 사용자 및 애플리케이션 모두에 대한 MQTT 프로토콜의 보안 분석을 단순화하여 스펙을 모르거나 정의하지 않고 프로토콜을 모호하게 할 수 있는 방법을 제공한다.

실험 결과는 fuzzed 패키지 당 허용 가능한 처리 시간을 제공하였다. 또한 도구가 삽입된 각 테스트 케이스 사이를 지나는 시간에 따라 다르게 작동하는 것으로 관찰하였다. 호스트 시스템의 CPU 소비를 최소 2%까지 줄일 수 있었다. CPU 소비를 제어할 수 있는 이러한 유연성 덕분에 스마트 시티와 같은 처리 능력이 낮은 장치가 있는 환경에서 이 도구를 사용할 수 있다. 이 도구는 MOQUETTE 또는 MOSQUITTO와 같이 광범위하게 사용되는 클라이언트의 취약성을 테스트하는데 사용되었으며 서비스 거부 및 브로커의 통신 재설정과 같은 문제가 보고되었다. 발견된 취약점은 도구의 효과를 확인한다.

프레임 워크에는 또한 몇가지 한계가 있었다. 가장 중요한 것은 오류 보고 및 탐지와 관련이 있다. 오류 검출은 디버거에서 테스트 할 응용 프로그램의 실행을 통해 수행된다. 분명히 효율성과 유용성을 개선하기 위해 자동화가 필요하다. 또 다른 중요한 제한 사항은 현재 프레임 워크가 MQTT 프로토콜 보안을 검증하는 데만 사용 가능하다는 것이다. 따라서 이 도구는 여러 프로토콜을 구현하는 IoT 아키텍처에는 효율적이지 않다.

앞서 언급한 한계점을 개선하기 위해 향후 프로젝트의 일환으로 IoT 장치가 사용하는 더 광범위한 네트워크 프로토콜을 검증할 수 있도록 프레임 워크를 확장할 것이다. 또한 이 도구를 처음으로 네트워크에 통합된 모든 요소에 대한 보안 분석을 수행하는 서비스로 사용할 가능성을 분석하고 있다. 이를 통해 인프라의 모든 구성 요소에 대해 최소 수준의 보안 및 안정성을 보장할 수 있다.