HW #1

CRC 구현 및 파라미터 세팅에 따른 성능분석

2025. 11. 21.

학과	소프트웨어학부
학번	20241560
이름	서준영
분반	(가)

■ 목차

■ 시뮬레이션 환경

구현 언어: python

시뮬레이션 컴퓨터: M2 MacBook Air (8 Core CPU / 10 Core GPU / 24GB)

모든 파일은 Github 에 업로드 되어있다. (https://github.com/jysuhr/CRC_Simulation.git)

교재에 있는 CRC 별 Polynomial 을 사용한다.

Name	*Polynomial*	*Used in*
CRC-8	x^8+x^2+x+1 **100000111**	ATM header
CRC-10	$x^{10}+x^9+x^5+x^4+x^2+1$ **11000110101**	ATM AAL
CRC-16	$x^{16}+x^{12}+x^5+1$ **10001000000100001**	HDLC
CRC-32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$ **10000010011000001000111011011011**	LANs

CRC 별 data word 의 길이와 에러 비트 비율을 공지에 따라 정의한다.

```
parameters = [
        # (1) CRC-8
        (CRC_TYPE.CRC8, 16), (CRC_TYPE.CRC8, 32), (CRC_TYPE.CRC8, 64), (CRC_TYPE.CRC8, 128),
        # (2) CRC-10
        (CRC_TYPE.CRC10, 32), (CRC_TYPE.CRC10, 64), (CRC_TYPE.CRC10, 128), (CRC_TYPE.CRC10, 256),
        # (3) CRC-16
        (CRC_TYPE.CRC16, 32), (CRC_TYPE.CRC16, 64), (CRC_TYPE.CRC16, 128), (CRC_TYPE.CRC16, 256),
        # (4) CRC-32
        (CRC_TYPE.CRC32, 64), (CRC_TYPE.CRC32, 128), (CRC_TYPE.CRC32, 256), (CRC_TYPE.CRC32, 512),
    ]
    error_rates = [0, 1, 10, 20, 50]
```

이중 반복문을 이용하여 다양한 파라미터의 조합을 시뮬레이션 한다.

```
# 시뮬레이션 조건
    for crc_type, data_word_length in parameters:
        for error_rate in error_rates:

            print(f"\n[RUN]: {crc_type.name} / Data:{data_word_length}bit / Error:{error_rate}%")
            stats = runSimulation(crc_type, data_word_length, error_rate)
            report_stats.append(stats)

    print("\n--- All Simulations Complete ---")
```

■ 시뮬레이션 과정

함수 구조

```
def generateDataWord(length):
```

- data word를 길이에 맞춰 랜덤하게 생성하는 함수

```
def generateErrorPattern(length, errorRate):
```

- 길이에 맞는 에러 비트 수를 계산하여 에러 패턴을 랜덤 생성하는 함수

```
def calculateCRC(CRCType: CRC_TYPE, dataWord):
```

- CRC type에 맞는 polinomial을 사용해서 Shift, AND, XOR Bitwise 연산으로 CRC 값을 계산 계산하는 함수

```
def makecodeWord(CRCType: CRC_TYPE, dataWord, crc):
```

- CRC type을 보고 CRC bit 길이만큼 dataWord를 Left Shift 시키고 crc 값과 OR 연산으로 결합하여 Code word를 만드는 함수

```
def runSimulation(
        crc_type: CRC_TYPE,
        data_word_length: int,
        error_rate: int,
        run_num = 4096):
```

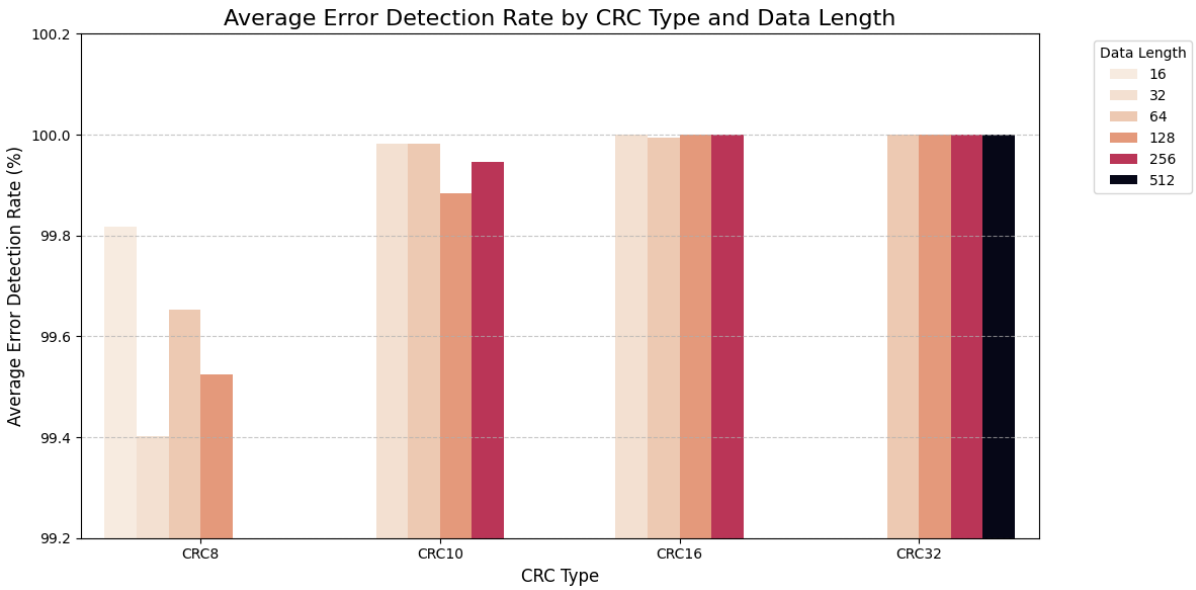
- 파라미터의 한 조합에 대하여 4096회 반복 시뮬레이션을 수행하는 함수
- 시뮬레이션 1회의 과정
  1. generateDataWord() , generateErrorPattern()을 호출한다.
  2. 타이머를 시작한다.
  3. calculateCRC()를 호출한다. – CRC 값 계산
  4. makecodeWord()를 호출한다. – code word 완성
  5. code word와 error pattern을 XOR 하여 Error를 주입한다.
  6. calculateCRC()를 호출한다. – Syndrome 계산
  7. 타이머를 종료한다.

  * '연산 딜레이'는 송신자가 CRC 를 계산하여 data word 를 만들고 수신자가 Syndrome 을 계산하는 과정까지이다.
- 각 파라미터 조합의 시뮬레이션 결과를 서로다른 CSV 파일에 저장
- 각 파라미터 조합의 평균을 Summary_Report_Statistics.csv 파일에 저장

■ 데이터 분석

- Google colab python을 이용해서 csv 파일을 분석 후 시각화했다.

1. CRC 타입별 평균 에러 감지 확률



CRC type과 Data Length에 따른 Error Detection 평균의 상관관계를 나타낸 Bar plot이다.

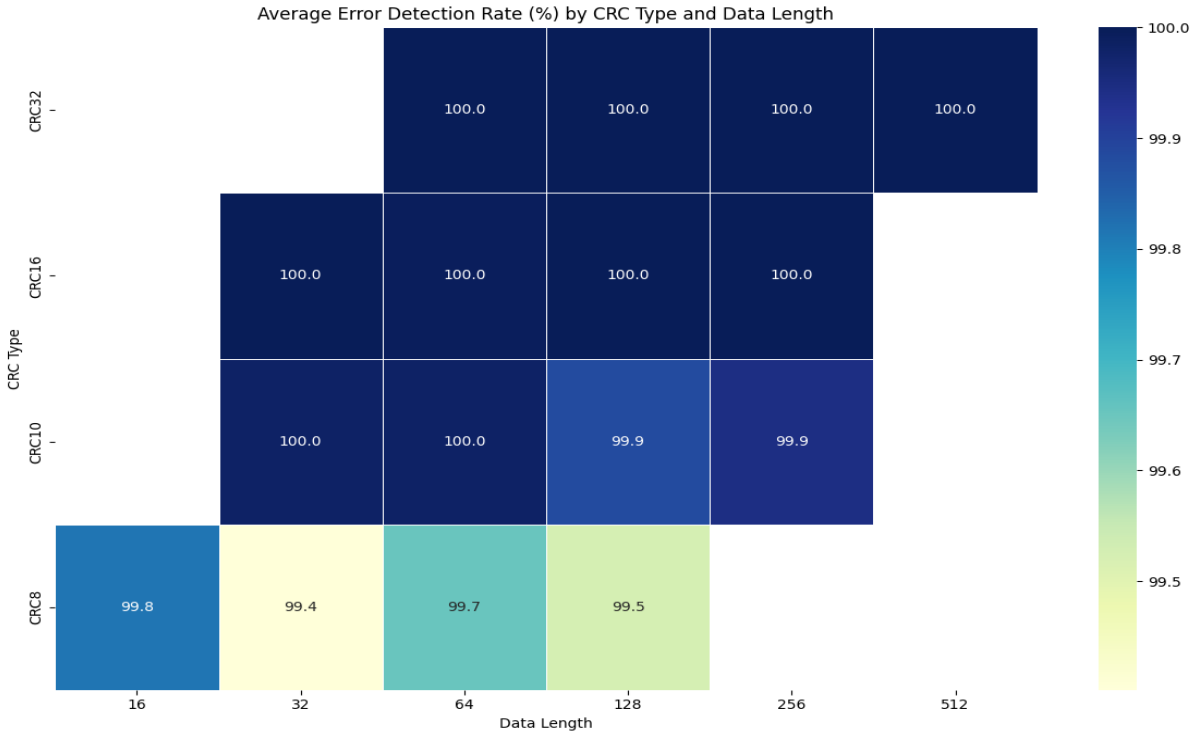
평균 에러 탐지율 (Average Error Detection Rate)은 CRC타입에 대해 모든 에러 비율에서 측정된 탐지율의 평균을 나타낸다. 한 조합당 4096회의 반복 실행 중 에러가 탐지된 횟수를 측정했고 계산 식은 다음과 같다.

$$D_e(\%) = Detection\ Rate_e(\%) = \frac{Errors\ Detected}{Errors\ Injected} \times 100$$

e: *에러 비트 비율*

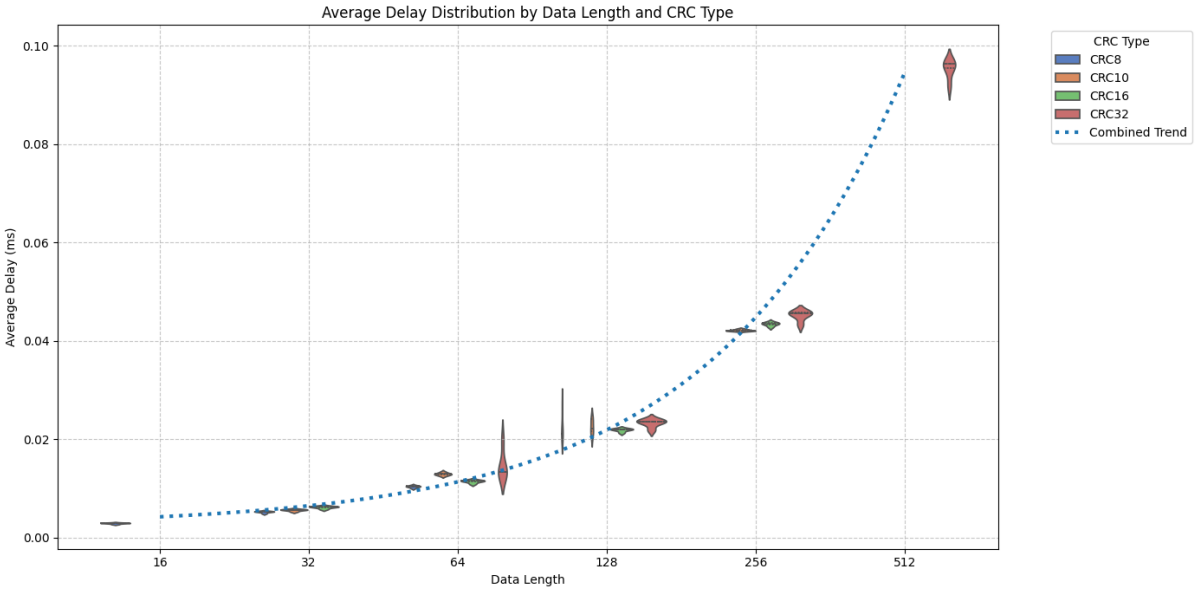
$$Average\ Error\ Detection\ Rate(\%) = \frac{D_1 + D_{10} + D_{20} + D_{50}}{4}\ (\%)$$

네 개의 CRC type은 모두 Average Detection Rate가 99.2%~100%에 분포하며 대부분 준수한 정확도를 보여준다. 상대적으로 비교하면 CRC-8, CRC-10, CRC-16, CRC-32 순으로 높은 정확성을 확인할 수 있다.



(▲ 위와 같은 데이터의 Hitmap)

2. CRC 타입별 평균 연산 딜레이

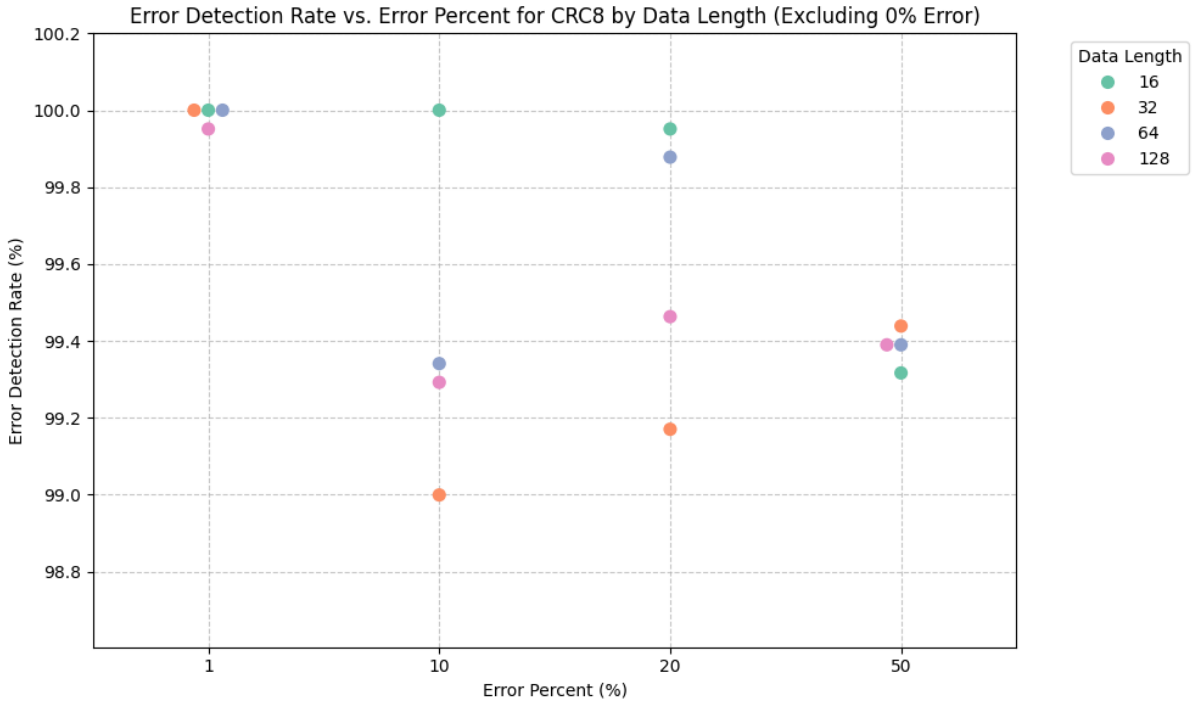


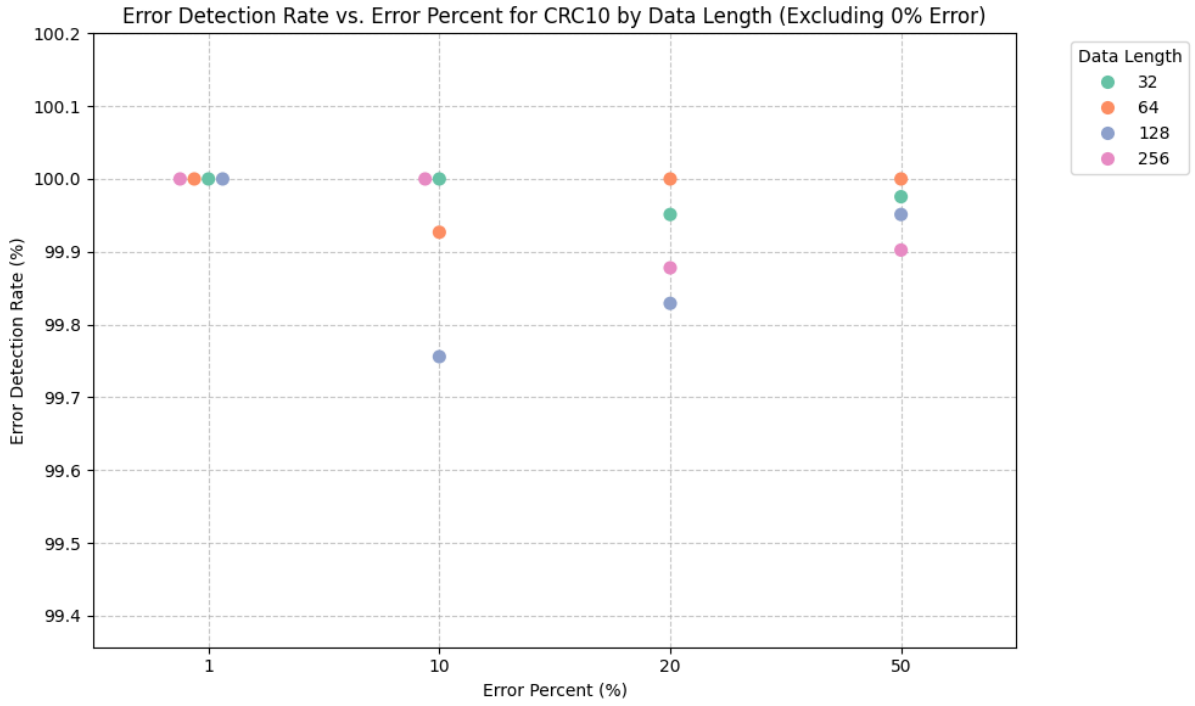
Data word 길이에 따른 연산 딜레이 Violin 분포도에 추세선을 그린 그래프이다.

가로축은 2배씩 증가하는 로그 스케일로 되어있다. 추세선 계산 결과 지수가 1.097로 거의 선형(x^1)에 가까우므로 Data word 길이에 따른 Average Delay는 선형적으로 증가함을 알 수 있다.

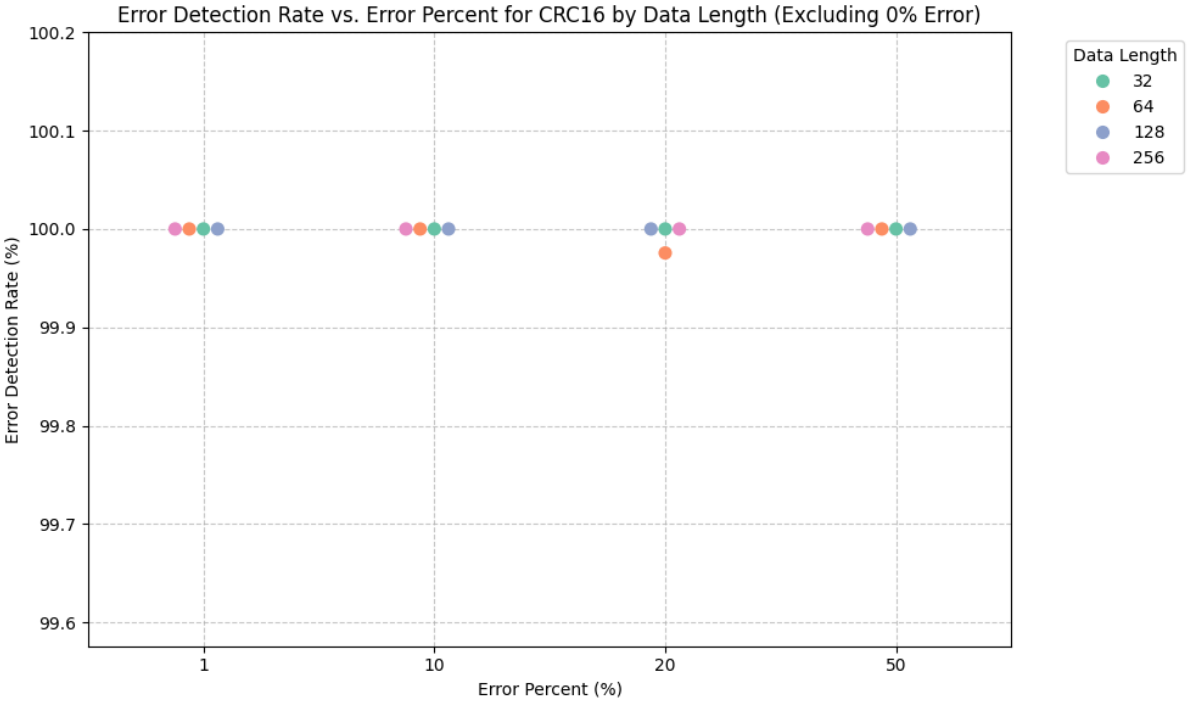
3. 에러 주입 비율, 데이터 길이 - 평균 에러 감지 확률

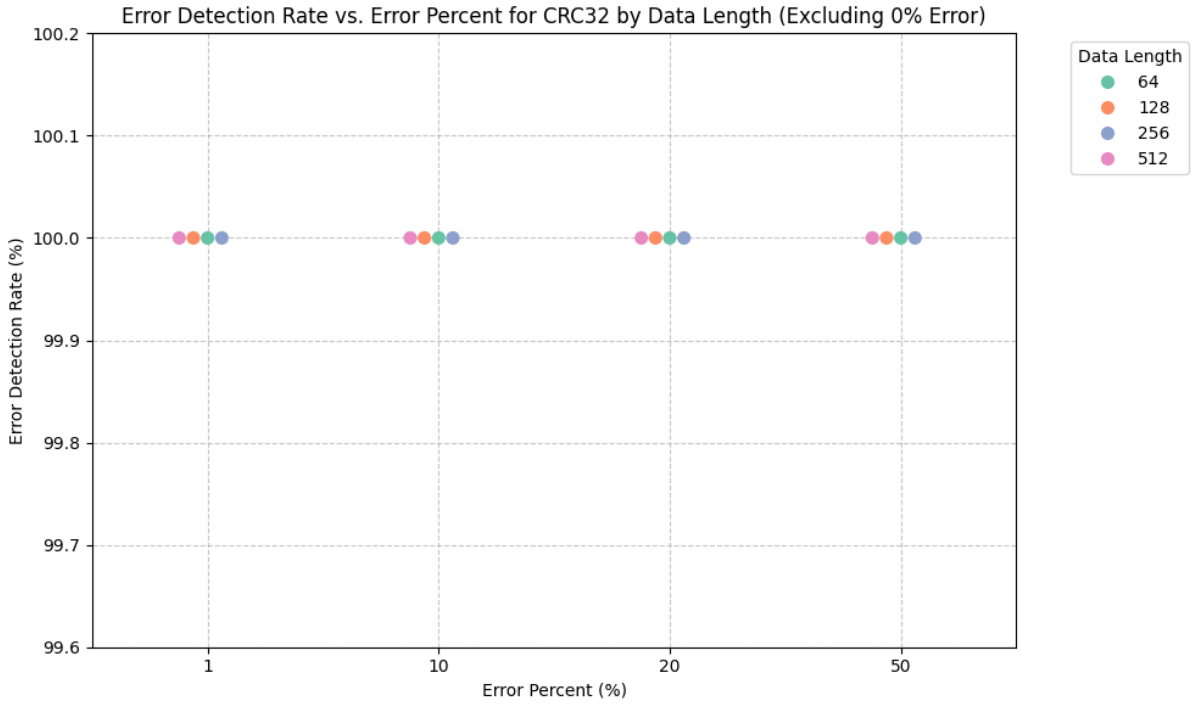
CRC type 별 데이터 길이에 따른 Error Detection 확률 그래프이다.





CRC-8과 CRC-16은 에러 비트 비율 1%에서는 100%에 가까운 Error Detection 확률이 나타난다. 에러 비트 비율이 증가함에 따라 Error Detection 확률은 감소하는 추세가 보이며, 데이터 길이가 길어질 때에도 마찬가지로 Error Detection 확률이 감소함이 확인된다.





CRC-16과 CRC-32에서는 모든 데이터 길이와 에러 비트 비율에 대해 모두 100%에 가까운 Error Detection 확률이 확인된다.

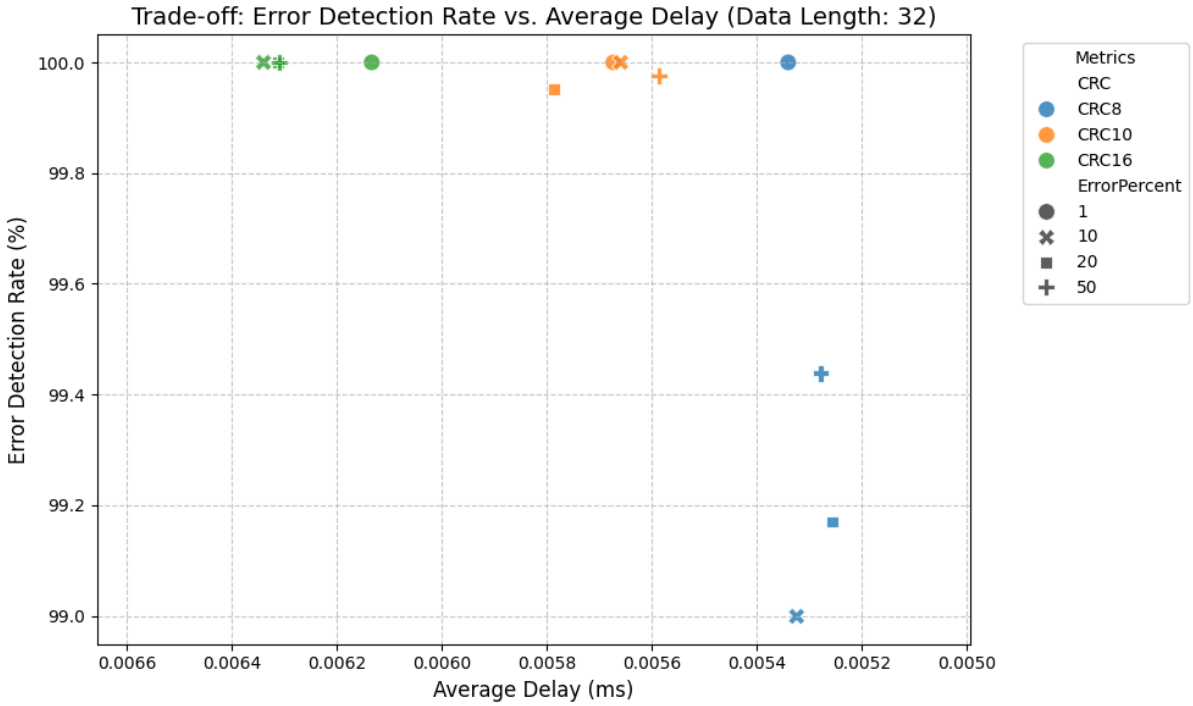
4. 파라미터 조합별 최적 CRC 분석

- 세로 축 : 평균 Error Detection Rate (%)

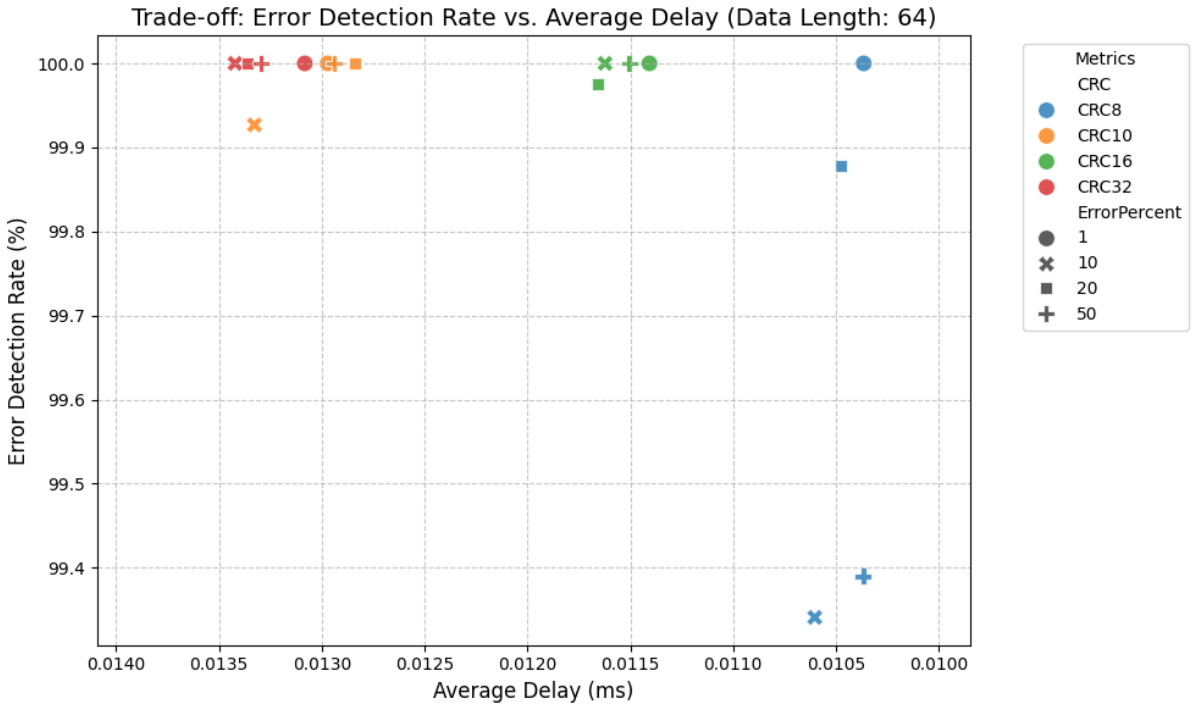
- 가로 축 : Average Delay (ms), 오른쪽으로 갈수록 짧아지게 설정했다.

- 오른쪽 위에 점이 있을 수록 성능이 좋은 것으로 평가한다.

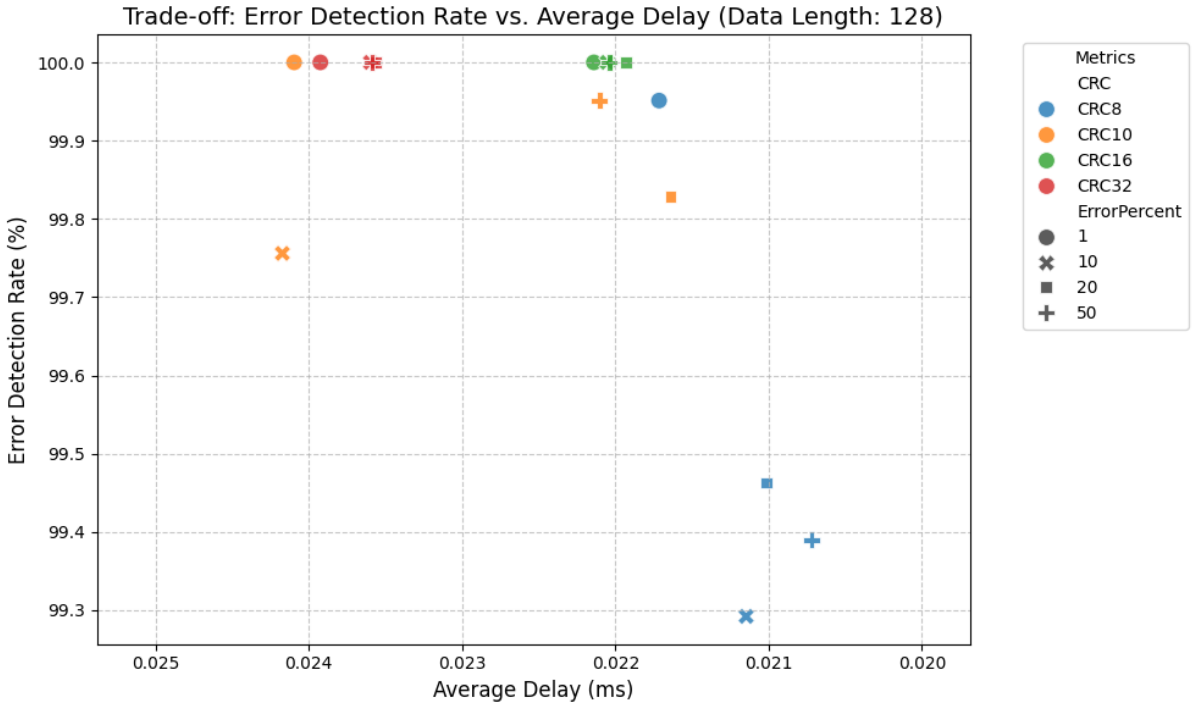
- Data length 16과 512는 하나의 CRC 종류만 시뮬레이션 했으므로 제외한다.



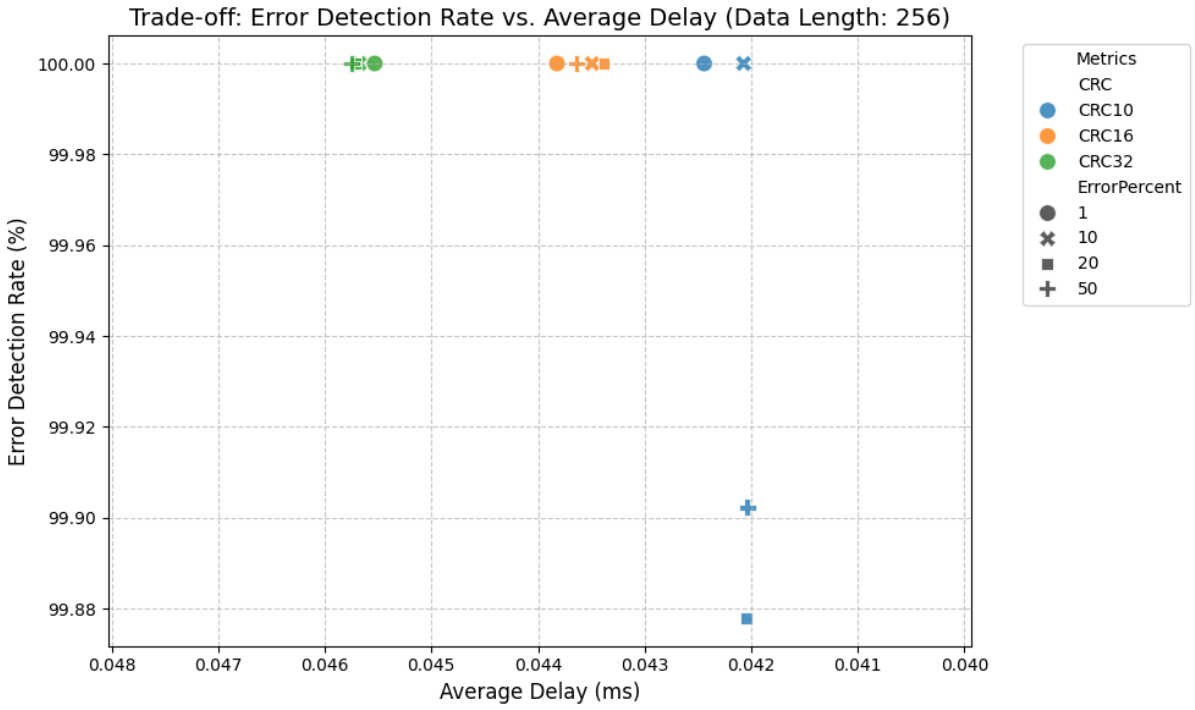
Data Word 길이 32 bit 에서는 CRC-8이 가장 짧은 딜레이를 기록했으며, CRC-10, CRC-16 순으로 연산시간이 길어진다. Error Detection 확률은 대체로 100%에 근접했으나 CRC-8에서는 에러 주입률에 따라 최대 1%p까지 하락했다.



Data Word 길이 64 bit 에서는 CRC-8이 가장 짧은 연산 딜레이를 보여주며, CRC의 길이가 길어질 수록 연산 딜레이가 길어진다. CRC 길이가 길어질 수록 Error Detection 확률은 100%에 근접함을 확인할 수 있다.



Data Word 길이 128 bit 에서는 CRC-8은 가장 적은 연산 딜레이를 보여주지만 상대적으로 Error Detection 확률이 떨어진다. CRC-16과 CRC-32는 Error Detection 확률이 100%에 근접하여 비슷하지만, 연산 딜레이는 CRC-16이 더 짧으므로 CRC-16이 더 효율적이라고 판단할 수 있다.



Data Word 길이 256 bit 에서는 CRC-10의 Error Detection 확률이 0.1% 범위 내에서 소폭 하락함이 나타난다. CRC-16과 CRC-32는 100%의 Error Detection 확률을 보여준다. 연산 시간이 짧아야 하는 환경에서는 CRC-10이 최적일 것으로 판단할 수 있다.

■ 결론

시뮬레이션 결과 CRC의 종류와 데이터 길이에 따른 성능 Trade-off 관계를 확인할 수 있었다.

에러 검축 성능 면에서 CRC의 비트 수가 증가할 수록 안정적인 검출률을 보였다. 특히 CRC-32에서는 모든 실험 조건에서 100%에 근접한 에러 검출 확률을 기록하며 신뢰성이 가장 높았다. 반면 CRC-8은 데이터 길이가 길어지거나 에러 주입 비율이 높아질 경우 검출률이 다소 하락하는 경향을 보였다.

연산 효율성 측면에서는 CRC 비트 수가 적을 수록 연산 지연 시간이 짧게 나타났다. CRC-8이 가장 빠른 연산 속도를 보였으며, CRC-32는 상대적으로 가장 긴 연산 시간이 소요되었다. 또한 데이터 길이가 증가함에 따라 연산 지연 시간은 선형적으로 증가함을 확인하였다.

짧은 데이터(32bit, 64bit) 전송 환경에서는 CRC-8과 CRC-10이 적절한 에러 검출 성능을 유지하면서 빠른 처리 속도를 보여 효율적이다. 그러나 128bit 이상의 긴 데이터를 전송하는 환경에서는 연산 시간이 다소 길어지더라도 CRC-16이상의 방식을 사용하는 것이 데이터 무결성을 보장하는데 유리하다고 판단한다.