It Doesn’t Have to Be So Hard: Efficient Symbolic Reasoning for CRCs

Vaibhav Sharma, Navid Emamdoost, Seonmo Kim,
Stephen McCamant
University of Minnesota,
Minneapolis, MN, USA

Motivation

- Cyclic Redundancy Check (CRC) is commonly used for error detection
- Not resistant to adversarial modification
  - WEP, SSHv1
- Sometimes used as an obstacle to symbolic execution
  - Jung et al., USENIX Security 2019
- Is analysis of CRC difficult?

Our Contribution

- It is not difficult to analyze CRC implementations
- Use symbolic execution to compute pre-image of CRC
- Explore two kinds of CRC implementations
  - update CRC once for every input bit
  - update CRC using lookup table (for every 8 bits in input)

Our Contribution

- It is not difficult to analyze CRC implementations
- Use symbolic execution to compute pre-image of CRC
- Explore two kinds of CRC implementations
  - update CRC once for every input bit
  - update CRC using lookup table (for every 8 bits in input)

CRC symbolic pre-image computation

```c
int main() {
  char sym_str[LEN];// set to symbolic bytes
  unsigned int sym_crc = crc(sym_str, LEN);
  char conc_str[LEN];
  srand(time(NULL));
  for (int i = 0; i < LEN; i++)
    conc_str[i] = (char) rand();
  if (sym_crc == crc(conc_str, LEN)) {
    printf("found the pre-image\n");
  }
  return 0;
}
```

Our Contribution

- It is not difficult to analyze CRC implementations
- Use symbolic execution to compute pre-image of CRC
- Explore two kinds of CRC implementations
  - update CRC once for every input bit
  - update CRC using lookup table (for every 8 bits in input)

Our Contribution

- It is not difficult to analyze CRC implementations
- Use symbolic execution to compute pre-image of CRC
- Explore two kinds of CRC implementations
    - update CRC once for every input bit
    - update CRC using lookup table (for every 8 bits in input)

Our Contribution

- It is not difficult to analyze CRC implementations
- Use symbolic execution to compute pre-image of CRC
- Explore two kinds of CRC implementations
  - update CRC once for every input bit
  - update CRC using lookup table (for every 8 bits in input)

Used by Jung et al. to defeat symbolic execution

CRC Implementation Type #1

```c
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;   crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

`message` points to symbolic bytes

CRC Implementation Type #1

```
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;   crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

CRC Implementation Type #1

```
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;  crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

CRC Implementation Type #1

```c
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;  crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

CRC Implementation Type #1

```
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;  crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

CRC Implementation Type #1

```c
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;   crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

CRC Implementation Type #1

```
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;  crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

- Both sides of branch feasible
- Executed once for every bit in message
- Causes path explosion
- Can be easily alleviated with path-merging

CRC Implementation Type #1

```
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;   crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

Summarize both sides of branch into single formula

```
crc=(int)(crc ^ byte) < 0
    ? (crc << 1) ^ 0x04C11DB7
    : crc << 1
```

CRC Implementation Type #1

```
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;   crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

- Summarize both sides of branch into single formula
- Write side-effects of summary into local variable (crc)
- Skip branching, jump to immediate post-dominator of branch instruction

Our Contribution

- It is not difficult to analyze CRC implementations
- Use symbolic execution to compute pre-image of CRC
- Explore two kinds of CRC implementations
    - update CRC once for every input bit
    - update CRC using lookup table (for every 8 bits in input)

Used to make CRC computation faster

CRC Implementation Type #2

```c
unsigned int cgc_crc32(char *buf, int len) {
  unsigned int c = 0xffffffff;
  int n;
  for (n = 0; n < len; n++)
    c = table[(c ^ buf[n]) & 0xff]
        ^ (c >> 8);
  return c ^ 0xffffffff;
}
```

CRC Implementation Type #2

```
unsigned int cgc_crc32(char *buf, int len) {
  unsigned int c = 0xffffffff;
  int n;
  for (n = 0; n < len; n++)
    c = table[(c ^ buf[n]) & 0xff]
        ^ (c >> 8);
  return c ^ 0xffffffff;
}
```

`buf` points to symbolic bytes

CRC Implementation Type #2

```
unsigned int cgc_crc32(char *buf, int len) {
  unsigned int c = 0xffffffff;
  int n;
  for (n = 0; n < len; n++)
    c = table[(c ^ buf[n]) & 0xff]
        ^ (c >> 8);
  return c ^ 0xffffffff;
}
```

CRC Implementation Type #2

```
unsigned int cgc_crc32(char *buf, int len) {
  unsigned int c = 0xffffffff;
  int n;
  for (n = 0; n < len; n++)
    c = table[(c ^ buf[n]) & 0xff]
        ^ (c >> 8);
  return c ^ 0xffffffff;
}
```

CRC Implementation Type #2

```
unsigned int cgc_crc32(char *buf, int len) {
  unsigned int c = 0xffffffff;
  int n;
  for (n = 0; n < len; n++)
    c = table[(c ^ buf[n]) & 0xff]
        ^ (c >> 8);
  return c ^ 0xffffffff;
}
```

CRC Implementation Type #2

```c
unsigned int cgc_crc32(char *buf, int len) {
  unsigned int c = 0xffffffff;
  int n;
  for (n = 0; n < len; n++)
    c = table[(c ^ buf[n]) & 0xff]
        ^ (c >> 8);
  return c ^ 0xffffffff;
}
```

- Concrete table lookup with symbolic index
    a. Branch for every entry
    b. Read table contents into If-Then-Else expression
    c. Use theory of arrays
    d. Use the structure of the table to summarize its contents

CRC Implementation Type #2

```
unsigned int cgc_crc32(char *buf, int len) {
  unsigned int c = 0xffffffff;
  int n;
  for (n = 0; n < len; n++)
    c = table[(c ^ buf[n]) & 0xff]
        ^ (c >> 8);
  return c ^ 0xffffffff;
}
```

- Concrete table lookup with symbolic index
  - ~~a. Branch for every entry~~
  - b. Read table contents into If-Then-Else expression
  - c. Use theory of arrays
  - d. Use the structure of the table to summarize its contents

CRC Implementation Type #2

```
unsigned int cgc_crc32(char *buf, int len) {
  unsigned int c = 0xffffffff;
  int n;
  for (n = 0; n < len; n++)
    c = table[(c ^ buf[n]) & 0xff]
        ^ (c >> 8);
  return c ^ 0xffffffff;
}
```

- Concrete table lookup with symbolic index
  1. ~~Branch for every entry~~
  2. Read table contents into If-Then-Else expression
  3. Use theory of arrays
  4. Use the structure of the table to summarize its contents

CRC Implementation Type #2

```
unsigned int cgc_crc32(char *buf, int len) {
  unsigned int c = 0xffffffff;
  int n;
  for (n = 0; n < len; n++)
    c = table[(c ^ buf[n]) & 0xff]
        ^ (c >> 8);
  return c ^ 0xffffffff;
}
```

- CRC lookup tables have special property
  - `T[i XOR j]=T[i] XOR T[j]`
- Compute all values in table using a *spine*
  - Table with 256 entries has 8 elements in spine
  - Elements at positions 1, 2, 4, 8, 16, 32, 64, 128, 256 form spine

Our Contribution

- It is not difficult to analyze CRC implementations
- Use symbolic execution to compute pre-image of CRC
- Explore two kinds of CRC implementations
  - update CRC once for every input bit
  - update CRC using lookup table (for every 8 bits in input)

Our Contribution

- It is not difficult to analyze CRC implementations
- Use symbolic execution to compute pre-image of CRC
- Explore two kinds of CRC implementations
    - update CRC once for every input bit
        - Use path-merging to alleviate path explosion
    - update CRC using lookup table (for every 8 bits in input)

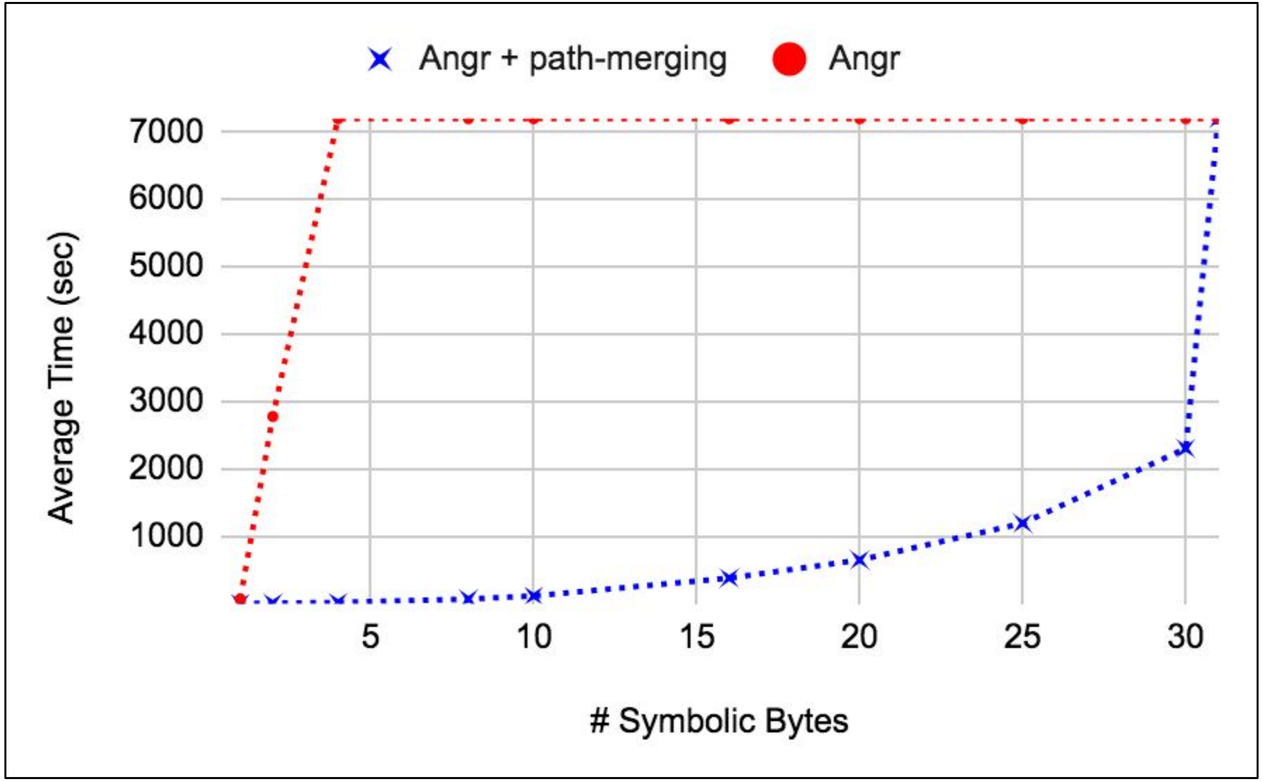
Our Contribution

- It is not difficult to analyze CRC implementations
- Use symbolic execution to compute pre-image of CRC
- Explore two kinds of CRC implementations
    - update CRC once for every input bit
        - Use path-merging to alleviate path explosion
    - update CRC using lookup table (for every 8 bits in input)
        - Summarize table using its structure

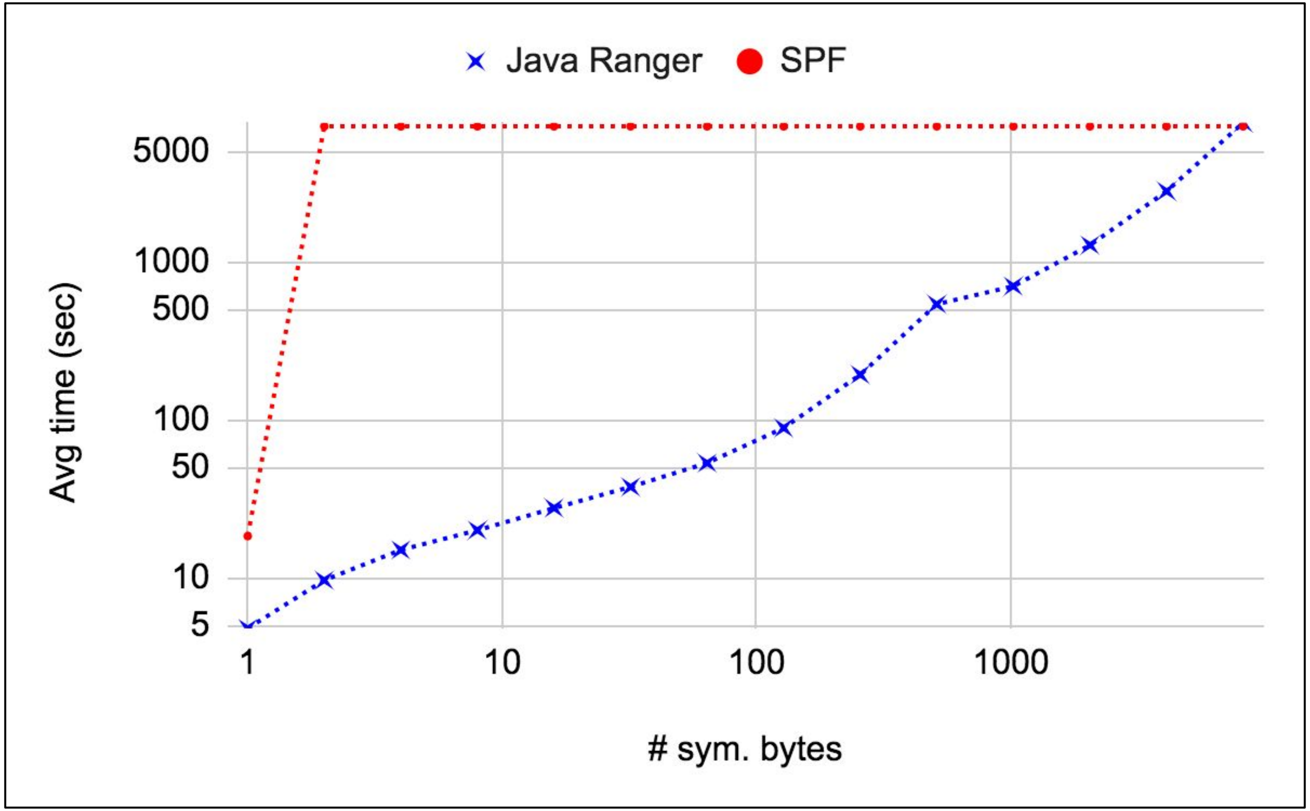
Evaluation

- Ran branching-based CRC implementation with and without path-merging
    - angr, Java Ranger (extension of Symbolic PathFinder)
- Ran table-based CRC implementation with FuzzBALL
    - ITE table treatment
    - theory-of-arrays support
    - GF(2)-linear table
- Used #sym input bytes ranging from 1 to 8192
- Time limit = 2 hours

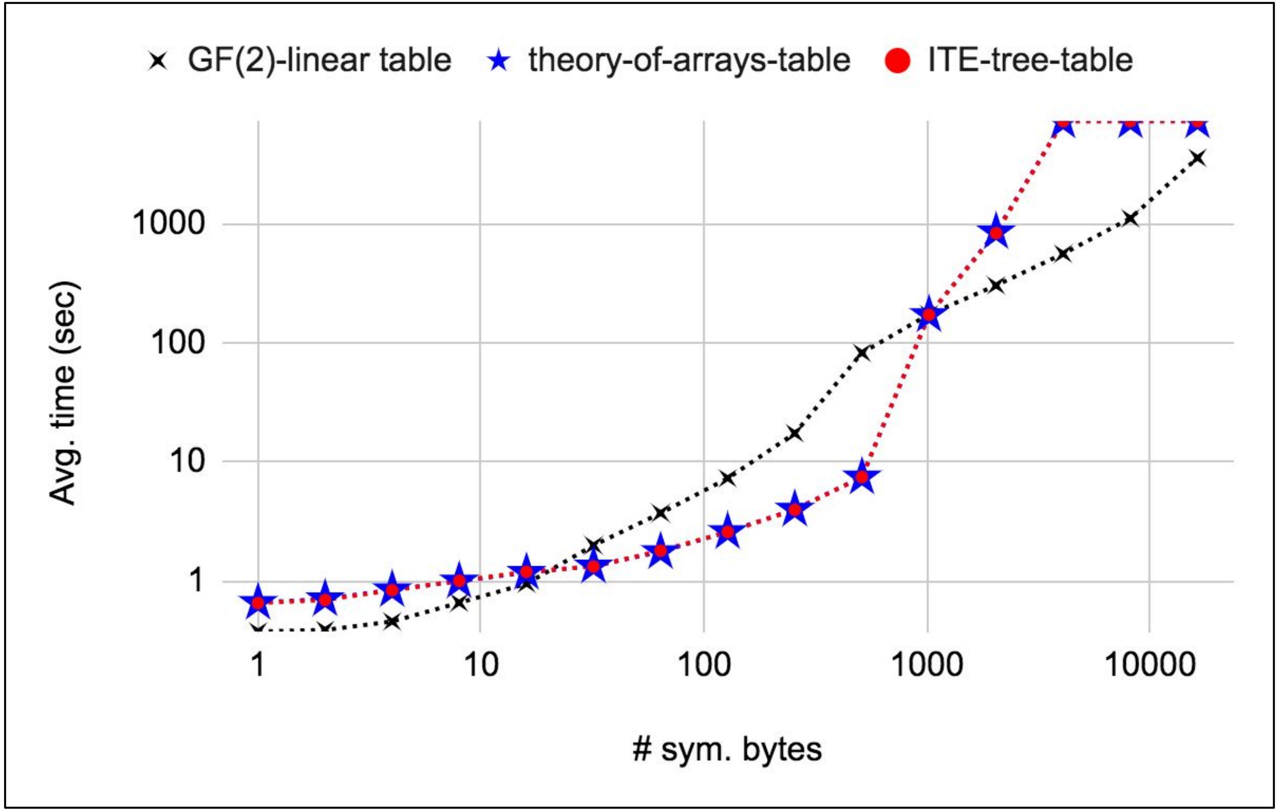
Evaluation: Branching-based CRC



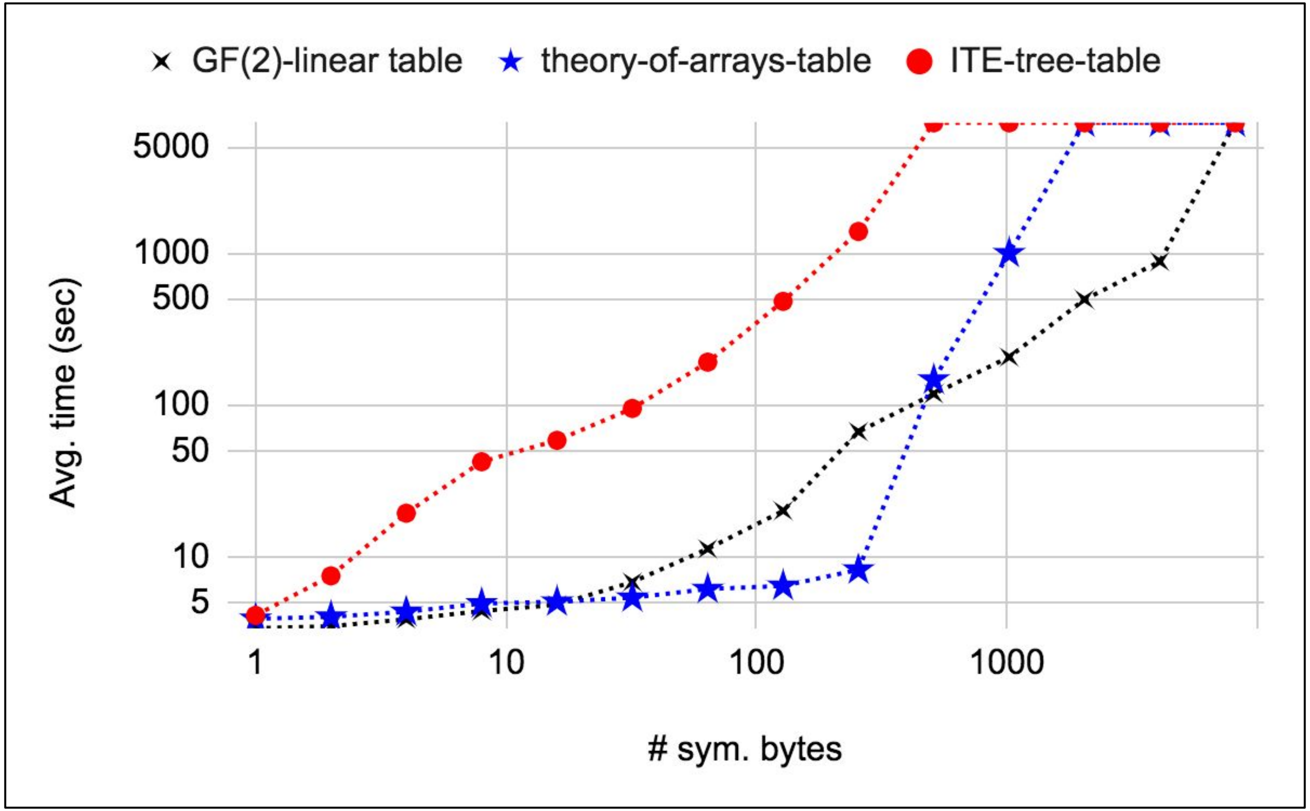
Evaluation: Branching-based CRC



Evaluation: Lookup-table-based CRC32



Evaluation: Lookup-table-based CRC64



Discussion

- Did not analyze in an end-to-end run of Jung et al.’s AntiHybrid technique
    - Used CRC as hash function because it is lightweight
- Found some variations of lookup-table-based CRC
    - Apache Hadoop CRC uses a 2048 entry table (8 tables, each with 256 entries)
- Related to
    - MultiSE (Sen et al., FSE 2015), Veritesting (Avgerinos et al., ICSE 2014)
    - Mayhem (Cha et al., IEEE S&P 2012) - bucketization with linear functions to create balanced index search tree

Conclusion

- Symbolic execution of CRC is not difficult
    - Anti-fuzzing techniques should use a different lightweight hash function
- Path-merging techniques accelerate branching-based CRC symbolic execution
- Proposed a new technique to improve scalability of symbolic CRC pre-image computation
    - Utilizes linear structure of CRC lookup tables

Questions

Questions

CRC Implementation Type #1

```c
unsigned int fuzzification_crc32(unsigned char *message) {
  int i, j;  unsigned int byte, crc;
  i = 0;   crc = 0xFFFFFFFF;
  while (message[i] != 0) {
    byte = reverse(message[i]);
    for (j = 0; j <= 7; j++) {
      if ((int)(crc ^ byte) < 0)
        crc = (crc << 1) ^ 0x04C11DB7;
      else crc = crc << 1;
      byte = byte << 1;
    }
    i = i + 1;
  }
  return reverse(~crc);
}
```

OK, because

CRC(A XOR B) =

CRC(A) XOR CRC(B)