Automatic Discovery and Synthesis of Checksum Algorithms from Binary Data Samples

Lauren Labell
Tufts University
lauren.labell@tufts.edu

Jared Chandler
Tufts University
jared.chandler@tufts.edu

Kathleen Fisher
Tufts University
kfisher@eecs.tufts.edu

ABSTRACT

Reverse engineering unknown binary message formats is an important part of security research. Error detecting codes such as checksums and Cyclic Redundancy Check codes (CRCs) are commonly added to messages as a guard against corrupt or untrusted input. Before an analyst can manufacture input for software which uses checksums they must discover the algorithm to calculate a valid checksum. To address this need, we have developed a program synthesis-based approach for detecting and reverse-engineering checksum algorithms automatically. Our approach takes a small set of binary messages as input and automatically returns a Python implementation of the checksum algorithm if one can be found.

Our approach first performs a search over the message space to identify the location of the checksum and then uses program synthesis to identify the operations performed on the message to compute the checksum. We return to the user runnable code to both calculate a checksum from a message and to validate a message according to the checksum algorithm. We generate unit tests, allowing the user to validate the synthesized checksum algorithm is correct with regard to the input messages.

We created the Tufts Checksum Corpus comprised of 12 checksum inference questions collected from posts on reverse engineering question and answer sites and 2 instances of common internet protocol checksums.

Our approach successfully synthesized the underlying checksum algorithms for 12 out of 14 cases in our test suite.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**; **Network security**; **Formal methods and theory of security**.

KEYWORDS

checksums, protocols, synthesis, reverse engineering, binary data

ACM Reference Format:
Lauren Labell, Jared Chandler, and Kathleen Fisher. 2020. Automatic Discovery and Synthesis of Checksum Algorithms from Binary Data Samples. In *15th Workshop on Programming Languages and Analysis for Security (PLAS '20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3411506.3417599

PLAS '20, November 13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8092-8/20/11…$15.00
https://doi.org/10.1145/3411506.3417599

1 INTRODUCTION AND BACKGROUND

Checksum inference is the task of recovering a checksum algorithm specification from example messages or a binary program. This task is performed by software developers, reverse engineers, and cybersecurity analysts. These users perform checksum inference while trying to understand proprietary data formats and legacy software, reverse engineering malware, and identifying unknown traffic on computer networks. A key motivation for checksum inference is the desire to manufacture messages acceptable to a system under test that uses a checksum as a method of input validation. By inferring a valid checksum, security tools like fuzzers [20] can increase the likelihood that the target program will accept the generated input.

Consider an analyst working on a black box system. Her goal is to explore the system for security weaknesses. She has been given a set of messages that have been observed being transmitted to the system. While her focus is vulnerabilities, she first has to learn how to construct messages the system will accept. For the analyst, then, checksum inference is an obstacle which she must overcome on the path to a larger goal. Specifically, she would like to know (1) is there is a checksum in the collection of messages, if so, (2) over what portion of the message is it calculated and (3) what algorithm is used to calculate it. An automated solution to this problem would allow her to focus on the more important work of exploring the system for vulnerabilities rather than trying to find relationships between bytes of different messages.

Checksums are one type of error detecting code [9]. Other types include cyclic redundancy check codes (CRCs) [14] and hash functions. Each type is distinct even if the terms are often conflated. A checksum algorithm repeatedly applies a binary operation across all the bytes being checked. In contrast, CRC algorithms are more complex and based on polynomial division. These algorithms treat the message as a large binary number, and divide it by a polynomial constant. The remainder of the division forms the check value. Although both checksums and CRCs provide error-detecting functionality, they are different approaches. In this work, we focus specifically on checksums.

The underlying checksum specification may be unavailable to the analyst for a number of reasons. The specification may have become lost in the case of a legacy system. The specification may be proprietary or incomplete.

Without a specification the analyst could perform static [21] and/or dynamic analysis [18] of the black box system to gain insight into the operation of the checksum algorithm, assuming she has access to the underlying program. She could attempt to induce the program to generate useful messages by making small alterations to input or system memory and looking for corresponding changes in output messages. As a last resort she can examine the messages themselves in hopes of gaining insight into the algorithm

producing the checksum. This manual approach can prove even more difficult if the presence and position of the checksum is unknown as well as which portions of the message are included in the checksum calculation. All of these steps are time consuming and labor intensive.

In this work, we present an automated approach to checksum inference which requires no input from the user beyond providing a set of binary messages. We decompose checksum inference into two sub-problems. First, there is the problem of identifying the contiguous portions of the message which comprise the checksum and the payload from which the checksum is calculated. Second, there is the problem of synthesizing the algorithm used to generate the checksum from the payload. Our approach searches for regions of messages and synthesizes concrete checksum algorithms consistent with the data. Our approach takes binary messages as input (Listing 1) and automatically produces runnable checksum algorithm code (Listing 2). We focus specifically on the problem of inferring checksum algorithms and assume only a set of binary messages from the same protocol using a common algorithm to calculate the checksum.

Listing 1: Example input messages provided by analyst

```
1 08009654A41700005EAB245C00044F8508090A0B0C0D0E0F101112131
  415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2F30
  31323334353637
2 00009E54A41700005EAB245C00044F8508090A0B0C0D0E0F101112131
  415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2F30
  31323334353637
3 08008239A41700015EAB245D0004639E08090A0B0C0D0E0F101112131
  415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2F30
  31323334353637
```

Listing 2: Example synthesized function output to analyst

```
1 def calculate_checksum(payload):
2   mask = 0xFFFF
3   checksum = 0
4   for element in payload:
5     checksum = onesComp(checksum,element)
6   checksum =  operator.invert(checksum)
7   return checksum & mask
```

The rest of the paper is structured as follows. In Section 2, we describe how we created the Tufts Checksum Corpus. We started by creating a corpus of 37 error detecting code inference problems from Reverse Engineering Stack Exchange [5] and Stack Overflow [13], each of which provided a set of binary messages. Of these 37 problems, we classified 12 as checksum algorithms and the rest as CRC algorithms, hash algorithms or unanswered problems. We also gathered an IPv4 [17] Header capture and an ICMP [16] capture, both of which use checksums. In total, we assembled a test suite of 14 checksum inference problem instances.

In Section 3, we describe our approach using a combination of search and program synthesis [8, 11]. Using insights gained from the examples in our test suite, we developed a parameterized model of the payload location, checksum location, and the checksum algorithm as well as an approach to search for checksum algorithms given a set of binary messages. We first perform a guided search for parameters describing payload and checksum location. Given a candidate payload and checksum we then attempt to synthesize a checksum algorithm using the payload as our desired input and the checksum as our desired output. Our approach exhaustively tests all parameterizations of the model and reports algorithms for which it correctly computes the checksum values for all messages.

In Section 4, we evaluate our approach on our test suite consisting of problem instances from the forums, an IPv4 protocol header capture, and an ICMP protocol capture. Out of these 14 problem instances, we found meaningful results for 12 of them. In addition, we report micro-benchmark data characterizing the performance of our approach for problem instances of various sizes.

In Section 4.1, we discuss our results. We discuss the two checksum algorithms which our algorithm was unable to synthesize as well as why our approach may find multiple valid parameterizations for a single problem instance, requiring the help of an analyst to sort through the results.

In Section 5, we describe other efforts to reverse engineer error detecting codes. These efforts are divided into two general approaches. First, approaches which assume access to the binary program calculating a checksum. Our program does not require a binary and instead requires only messages captured during transmission. Second, approaches which require only example messages containing an error detecting code somewhere within the message. Our approach differs from these in that we focus specifically on the general space of checksum algorithms and return runnable code rather than searching for a message format description among a small set of known checksum implementations.

Our technique shows promise on the small number of samples we found and we hope to improve and extend it as we collect more samples. We explore the next steps of this work in section 6.

We summarize the contribution of our work as follows:

- We create a test suite of checksum/CRC reverse engineering problem instances from internet forums and existing internet protocols.
- We develop a program synthesis approach which takes binary messages as input and automatically produces runnable checksum algorithm code as output to the analyst.
- We evaluate our approach on our test suite and synthetic micro-benchmark data, showing that we correctly infer the checksum in 12 out of 14 cases.

We plan on releasing both our implementation and the Tufts Checksum Corpus online in the near future [1].

2 THE TUFTS CHECKSUM CORPUS

To construct the Tufts Checksum Corpus, we first found error detecting code reverse engineering problem instances posted on the question and answer internet sites Reverse Engineering Stack Exchange [5] and Stack Overflow [13]. To find relevant posts we searched the sites using the keywords "CRC" and "checksum". We collected problem instances in which users provided a set of binary messages along with varying degrees of background information such as the type of program generating the messages, error-detecting code location and width, and information about other fields in the packet. We collected a total of 37 problem instances and classified each according to the algorithm used to calculate the error-detecting code. Table 1 shows the results: 14 used CRC

[1] https://github.com/laurenlabell/checksum_finder

Table 1: Frequency of Error Detecting Algorithms

Type of Algorithm	Number of Samples
CRC	14
Checksum	12
Hash	1
Unknown	10

Table 2: The Tufts Checksum Corpus

Source	Number of Examples
Reverse Engineering Site Posts	12
IPv4 Header	1
ICMP	1

algorithms, 12 used checksum algorithms, 1 used a hash code, and 10 were unknown.

Out of these different classes of algorithms, we chose to focus specifically on checksums because many people on the question and answer sites were trying to solve checksum problems manually. While the reverse engineers on these websites frequently used tools such as CRC RevEng to reverse CRC algorithms, no similar tool was used to reverse engineer checksum algorithms, problems that made up a significant number of posts on the forums.

Checksum function inference is often a secondary aspect of the more general problem of message format inference [10, 12]. Message reverse engineers are primarily concerned with other aspects of the message format, but they hit a roadblock when required to produce a correct checksum. For example, when they inject messages protected by a checksum into a system under test, they must correctly calculate the checksum. Messages without a valid checksum won't be accepted, thus producing a bottleneck and preventing progress in the inference of other fields. The people posting queries to the forums were not checksum experts, but sought expert understanding to overcome this challenge.

From these posts, we identified four challenges in reverse engineering checksums:

(1) Identifying the portion of the message protected by the checksum, *i.e.*, the payload portion of the message.
(2) Identifying the portion of the message comprising the checksum itself.
(3) Discovering the specifics of the checksum algorithm that takes the payload as input and returns the checksum as output.
(4) Reporting these findings to the user in a way that does not assume extensive background knowledge.

Many of the checksum examples employed similar design patterns that we generalize into a formal model.

To form the Tufts Checksum Corpus, which we summarize in Table 2, we started with the 12 checksum problem instances from the two reverse engineering forums. We then added two more problem instances that contain 16-bit checksums: IPv4 message headers and ICMP messages. The complete test suite can be found in the appendix.

3 APPROACH

Our approach to the problem of inferring checksums uses a combination of search and program synthesis. Figure 1 shows an overview of the algorithm. We first perform a guided search for parameters describing the payload (the subsection of the message used to calculate the checksum) and checksum locations. Given a candidate payload and checksum we perform an enumerative search to synthesize a checksum algorithm using the payload as input and the checksum as output. This approach reflects the assumption that checksum algorithms are agnostic to the size and location of the payload and the checksum.

Our approach uses a parameterized model with required parameters *Width*, *mStart*, *mEnd*, *mCheck*, and `FOLDOP`, and with optional parameters `UFINOP`, `BFINOP`, and `MAGIC`. Parameters in *italics* characterize the location of the payload and checksum in the message. Parameters in `teletype` characterize the checksum algorithm. Reverse engineering a checksum algorithm involves searching for parameters such that for all messages in a problem instance, the "sum" of the subarray delineated by *mStart* and *mEnd* is equal to the message at the index *mCheck*, where the operation used to calculate the "sum" is determined by the parameter `FOLDOP` and can be a function drawn from `addition`, `xor`, `subtraction` or `onesComplement`. The finalizing operation can be either a unary operation `UFINOP` or a binary operation `BFINOP` that combines the sum with a magic value `MAGIC`. The parameter *Width* controls whether the indexing is based on 8- or 16-bit quantities. For the case of a `FOLDOP` of `addition` and a binary finalizer:

$$\exists mStart, mCheck, mEnd \in validIndices \text{ such that:}$$
$$\forall m \in msgs,$$
$$\texttt{BFINOP}\left(\left(\sum_{i=mStart}^{mEnd} m[i]\right), \texttt{MAGIC}\right) = m[mCheck]$$

We divide our approach into two phases: (1) a search over the message space in which we explore all combinations of *mStart*, *mCheck* and *mEnd* parameters and (2) the synthesis of a checksum algorithm that identifies the fold and final operations. We assume the user has specified the value of the *Width* parameter.

As a running example we consider messages from the Tufts Checksum Corpus problem 4, shown in Listing 3.

Listing 3: Example messages from the Tufts Checksum Corpus problem instance 4

```
806FA30102B00818
806FA30112800878
1003A30001004006729E99940012120B
1003A30001003007709C98940012121F
1003A30001003806739C9B9400121202
```

3.1 Message Region Search

Developers use checksums in packets being transmitted through noisy channels, in which bit errors are relatively likely to occur. Therefore, a checksum algorithm cannot rely on the correctness of the message itself. Instead, the algorithm must operate at some known offset from the start or end of the message, independent of the data being transmitted. Additionally, assuming all messages use the same algorithm, the algorithm must be valid for every message

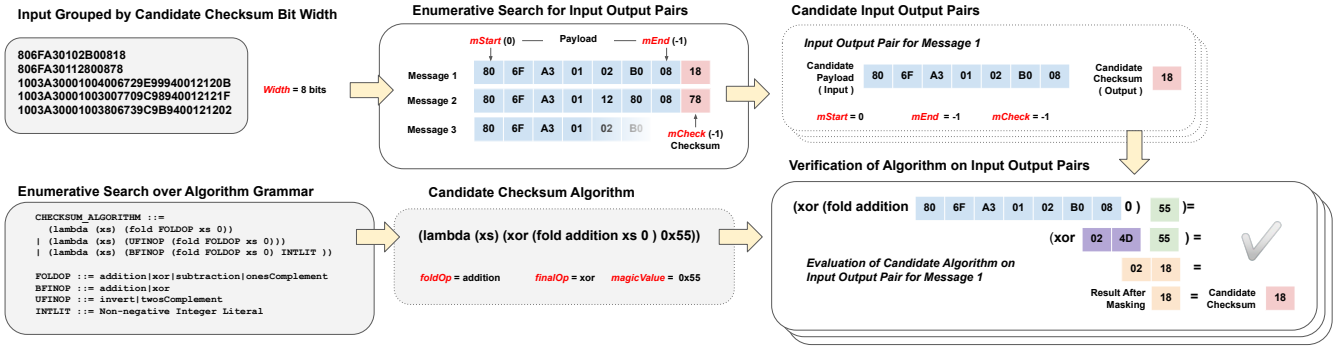

Figure 1: Overview of our combined search and program synthesis approach.

regardless of length. We use these intuitions to develop a method of searching the message space for the *mStart*, *mEnd*, and *mCheck* parameters.

We begin by dividing each message into an array of *Width*-sized integers. If a message cannot be divided evenly into *Width*-sized integers, we pad with zeros. The parameters *mStart* and *mEnd* delimit the payload, which we define as the subarray used as input to the checksum algorithm. In most problem instances, the payload forms a contiguous set of bytes; however, we found examples in which the checksum was located within the payload. Our approach handles this case by zeroing out the bits in the checksum location before computing the checksum. Zero is an identity for the fold operations (`addition`, `xor`, `subtraction`, and `onesComplement`) so by replacing the checksum with zero we remove it from the calculation. One limitation of our work is that we cannot find disjoint payload regions separated by anything other than the checksum field. The parameter *mCheck* represents the index of the checksum in the array. The checksum has a size of *Width* bits.

The set *validIndices* captures our rules for legal values for parameters *mStart*, *mEnd*, and *mCheck*. Indices that are valid for longer messages may be invalid for shorter ones because they exceed the bounds of the shorter messages. Therefore, we use the intersection of valid indices across all messages. Because valid indices for the shortest message will be valid for all other messages but not vice versa, we only explore valid indices of the shortest message. For example, the Tufts Checksum Corpus problem 4 has 8- and 16-byte messages. Therefore, *mStart* = 0, *mEnd* = -1 and *mCheck* = 0 is a valid parameterization for both message lengths. However, although *mStart* = 10, *mEnd* = -1 and *mCheck* = 0 is valid for 16-byte messages, it is not valid for 8-byte ones because *mStart* is out of range.

Index *mStart* begins relative to the start of every message and has the range [0, *minLen*), where *minLen* is the length of the shortest message. The payload subarray extends up to but does not include *mEnd*, which we calculate relative to the end of each message. We use the Python convention of negative indexing for *mEnd* and define a *mEnd* value of 0 to signify the end of the array. For each value of *mStart*, we calculate a range of *mEnd* values to try: [*mStart* - *minLen* + 1, 1). Figure 2 shows a solution for problem 4 from

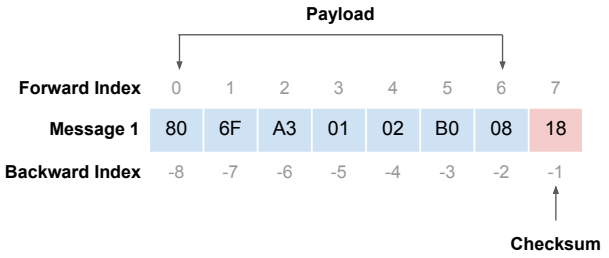

Figure 2: The Tufts Checksum Corpus problem 4 has the solution: *Width* = 8, *mStart* = 0, *mEnd* = -1, and *mCheck* = -1. Payload bytes are shown in blue, while the checksum is shown in red.

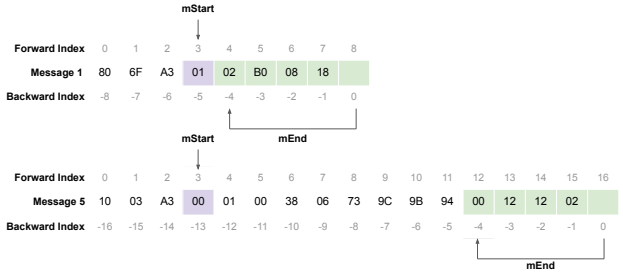

Figure 3: Valid values for *mEnd* (shown in green) of variable-length messages, given a particular *mStart* value (shown in purple). For problem 4, *minLen* is 8 so if we let *mStart* = 3, *mEnd* values are in the range [3 - 8 + 1, 1) = [-4, 1).

the Tufts Checksum Corpus, while Figure 3 shows valid values for *mEnd* given a particular *mStart* value.

To determine valid values for *mCheck* we consider the lengths of the messages. If all of the messages are the same length, we calculate *mCheck* relative to the beginning of the messages: [0, *minLen*). However, if the set includes messages of different lengths, we must also check indices relative to the end of the array: [-*minLen*,

Table 3: Message Space Parameters

Parameter	Description
Width	The width of the algorithm stride expressed in bits. We divide the messages into units of *Width* bits and the checksum is Width bits.
mStart	An index relative to the start of a message indicating the start of the payload.
mEnd	An index relative to the end of the message indicating the end of the payload. The payload extends up to but does not include *mEnd*.
mCheck	Index of the checksum relative either to the start (positive number) or the end of the message (negative number).

minLen). Because problem 4 includes messages of different lengths, we use the range [-8, 8).

We summarize the message space parameters in Table 3.

3.2 Checksum Algorithm Synthesis

Given a candidate payload and checksum described by the parameters *mStart*, *mEnd* and *mCheck*, we perform an enumerative search to synthesize a checksum algorithm using the payload as our input and the checksum as our output. In this section, we describe the grammar we use to model a checksum algorithm and provide an example demonstrating the verification of a checksum algorithm on a message.

```
CHECKSUM_ALGORITHM ::=
  (lambda (xs) (fold FOLDOP xs 0))
| (lambda (xs) (UFINOP (fold FOLDOP xs 0)))
| (lambda (xs) (BFINOP (fold FOLDOP xs 0) MAGIC))

FOLDOP ::= addition|xor|subtraction|onesComplement
UFINOP ::= invert|twosComplement
BFINOP ::= addition|xor
MAGIC  ::= Non-negative Integer Literal
```

Figure 4: Formal model of checksum algorithms.

We perform enumerative search bounded by the algorithm grammar shown in Figure 4 to synthesize checksum algorithms. We base the design of this grammar on the problem instances collected from Reverse Engineering Stack Exchange and Stack Overflow. Checksum algorithms consist of a fold operation `FOLDOP` singularly or in combination with either a binary final operation `BFINOP` and magic value `MAGIC` or a unary final operation `UFINOP`. For each payload input and checksum output, we perform a brute force search over all checksum algorithms. We verify a checksum algorithm by checking if the synthesized algorithm produces the checksum as output given the payload as input.

For the Tufts Checksum Corpus problem instance 4, we infer the following checksum algorithm:

Table 4: Derived parameterization for problem 4

Parameter	Value
Width	8
mStart	0
mEnd	-1
mCheck	-1
algorithm	`(lambda (xs)` `(xor (fold addition xs 0) 0x55))`

```
(lambda(xs)
       (xor (fold addition xs 0 ) 0x55))
```

which we call `checkSumA4`.

While the values of *mStart*, *mEnd*, and *mCheck* describe the location of the payload, the checksum algorithm indicates the operations performed on the payload to calculate the checksum. We refer to the combination of the description of the message space and a synthesized algorithm as the parameterization for a problem instance. Table 4 shows the parameterization for problem instance 4.

As an example of the application of `checksum4A` to message 1 from problem 4 (`806FA30102B00818`), we first apply the fold operation of `addition` over the payload.

```
foldResult = 0x80 + 0x6F + 0xA3 + 0x01 +
             0x02 + 0xB0 + 0x08 = 0x24D
```

We perform an optional final operation at the end, which may either be a unary operation or a binary operation requiring a non-negative integer known as a magic value. Although our approach is based on exhaustively testing all parameterizations of the model, we do not need to search for the magic value because we can calculate it in constant time. Specifically, we use known inverse functions to checksum algorithm final operations to identify a magic value for the first message and then verify that this magic value holds for all messages. We can therefore avoid searching for magic values in the range $[0, 2^{Width})$ and the runtime will not grow with the *Width*. This method allows our approach to be extended to larger widths (such as 32 bits) without compromising runtime. For problem 4, we `xor` the foldResult with a magic value of `0x55`.

```
finalResult = 0x24D ^ 0x55 = 0x218
```

Finally, we discard all but the least significant *Width* bits:

```
m[mCheck] = 0x218 & 0xFF = 0x18
```

3.3 Design Choices and Optimizations

In this section, we describe steps we take to present better inferred descriptions to the user, which involves prioritizing results that are most relevant to the user. We also discuss ways we optimize performance on long messages, including the use of dynamic programming.

We use a number of heuristics to present better inferred descriptions to the user. First, we expect checksums to exhibit a high degree of Shannon entropy [19] as we expect good checksum algorithms to distribute the checksum values over the entire expressive space of the checksum field. Therefore, to prevent extraneous results, our approach skips parameterizations in which the value at the

checksum index is constant for all messages. In addition, we sort the candidate checksum indices by Shannon entropy and explore indices with the highest Shannon entropy first, thus exploring parameterizations which are most likely to yield meaningful results first.

Additionally, the payload usually includes most if not all of the bytes in the message. Parameter *mStart* counts up from 0 and parameter *mEnd* counts down from 0, allowing our method to prioritize exploring payloads which include the entire message before exploring smaller portions iteratively.

We have also taken a number of measures to ensure that our approach runs efficiently on average data. First, if messages in a problem instance are of variable lengths, our approach checks the shortest messages first. Because the synthesis portion prunes algorithms by finding contradictions, not all messages need to be checked to prove that a certain parameterization does not work. If a parameterization does not work on one of the messages, then it is not valid. Checking the shorter messages first allows us to find these contradictions quicker.

Finally, while we use the simple search described above for short messages, we shift to a dynamic programming approach for messages over 50 bytes to speed up the runtime. The dynamic programming approach helps reduce the work involved in searching over the message space for parameters *mStart* and *mEnd*, enabling us to quickly verify a synthesized algorithm for many different candidate payloads. Using this approach, we compute the result of a fold operation for a candidate payload by utilizing the stored result for a smaller subsection of the payload. For example, to compute the result of a fold operation for a parameterization in which *mStart* = 0 and *mEnd* = 4, we could naively apply the operation across each index in the payload. However, if instead we stored the result of the same operation applied to the subproblem where *mStart* = 0 and *mEnd* = 3, we can apply the operation once to the result of this subproblem and the value at index 4 to compute the desired value. Overall, this approach saves us from performing unnecessary repetitive work.

4 EVALUATION

We evaluated our approach on the problems in the Tufts Checksum Corpus. The problems in the corpus from the Reverse Engineering forum allowed us to see if we could produce the same answer as a human. The IPV4 and ICMP message headers allowed us to evaluate our approach on instances in the wild. Finally, we also evaluated our approach on micro-benchmark data to let us characterize performance for conditions of message size and checksum width. We evaluated these micro-benchmarks using both the original and the dynamic programming approach to understand their impact on performance.

We ran our experiments on a Dell PowerEdge R815 server with 4 16-core 2.4 gigahertz AMD Opteron 6278 processors giving 64 cores total and 128 gigabytes of physical memory. Each experiment was run single threaded limiting the program to a single core.

Currently, we have evaluated our approach only on algorithms with *Width* values of 8 and 16 bits. We plan to extend our approach to larger widths in future work.

4.1 Discussion of Results: What did our method do?

We ran our approach on the 14 problems in the Tufts Checksum Corpus and compared the output of our approach with ground truth, obtained either from a forum posting or from a formal specification. For the examples from IPv4 headers and ICMP messages, our approach reported checksum algorithms consistent with the formal specification for each protocol. For the forum examples, our approach synthesized the ground truth checksum in 10 out of the 12 cases. Table 7 in Appendix A provides the ground-truth checksum algorithm, parameter values and total number of inferred checksum algorithms for each problem instance in the corpus. We note that the checksum problem instances from the forums lack formal specifications. Therefore, the algorithms communicated in the answers are not authoritative. Humans inferred solutions consistent with the data, but other solutions could exist. Samples messages for each problem instance are included in Appendix A.1.

We failed to find the checksum for the first of the remaining two examples because it computed the checksum modulo 257 instead of 256. Usually, developers use the modulo operator to ensure that the checksum fits in the correct number of bits; computing the sum modulo 257 does not guarantee that the checksum fits in a byte. We failed to find the checksum for the second example because it performed three final operations: `((sum(messageBytes) - 0x58) % 0xFF) + 1`. We can express nearly all of the checksum samples collected in terms of a single final operation so we chose not to support so many final operations. Nevertheless, our approach can be helpful even in these degenerate examples. Because we formally describe the scope of our approach, even if we do not find any results, we have ruled out the parameterizations explored in the search.

Our approach often finds many results for a single problem instance, obscuring the task of analyzing the output. Therefore, an analyst may be required to sort through the results, all of which are consistent with the data. For the 12 problem instances in our corpus, the number of results varied from 2 to 161. We attribute this result to two basic causes. First, a single checksum location may have multiple equivalent parameterizations. These multiple parameterizations are possible for the following reasons:

(1) Multiple algorithms are mathematically equivalent:
   - A final operation of `addition` or `xor` with a magic value of `0` is equivalent to no final operation.
   - A fold operation of `addition` with a final operation of `twosComplement` is equivalent to a fold operation of `subtract` with no final operation.
   - A final operation of `xor` with a magic value of all 1 bits is equivalent to a final operation of `invert`.
(2) We replace the checksum with zero in order to handle the case in which the checksum is located inside the payload. However, because we replace it with zero, we find two solutions for examples in which the checksum is located outside, but directly adjacent to the payload. The solutions will share the same parameters except that one includes the checksum in the payload as either the first or last index and the other does not.

Table 5: Derived Parameterization for `0x01 0x02 0x03 0xFA`

Parameter	Value
Width	8
mStart	0
mEnd	0
mCheck	3
algorithm	`(lambda (xs) (fold subtraction xs 0))`

(3) Multiple valid parameterizations for a given checksum index exist and depend on which bytes they include or exclude. For example, if all messages have regions of constant bytes we can only speculate whether or not to include those bytes in the calculation. Including or excluding them may only change the final operation's magic value.

Second, certain checksum algorithms exhibit mathematical properties which lead our approach to find checksums in many locations, despite the existence of only one designated checksum field. We observe this situation in the following algorithms:

- `(lambda (xs) (fold subtraction xs 0 ))`
- `(lambda (xs) (invert (fold onesComplement xs 0))`

In both these cases, the checksum can be verified with a simple calculation over both the payload and the checksum. For example, the checksum algorithm `(lambda (xs) (fold subtraction xs 0 ))` can be verified by taking the sum of the payload and the checksum. If this calculation adds up to 0 after discarding all but the least significant *Width* bits, the check succeeds. For example, consider the parameterization for example message `0x01 0x02 0x03 0xFA` shown in Table 5:

We can first verify the parameterization as written:

```
mCheck = 3:  (-0x01 -0x02 -0x03 -0x00) & 0xFF =
                                -0x06  & 0xFF =
                                        0xFA = m[3]
```

However, we also note that *mCheck* can take on any valid index and we still obtain a valid parameterization:

```
mCheck = 2: (-0x01 -0x02 -0x00 -0xFA) & 0xFF =
                               -0xFD  & 0xFF =
                                       0x03 = m[2]

mCheck = 1: (-0x01 -0x00 -0x03 -0xFA) & 0xFF =
                               -0xFE  & 0xFF =
                                       0x02 = m[1]

mCheck = 0: (-0x00 -0x02 -0x03 -0xFA) & 0xFF =
                               -0xFF  & 0xFF =
                                       0x01 = m[0]
```

Because our approach finds checksums at many indices in these cases, we can infer the true checksum index by considering the Shannon entropies of each index.

4.2 Discussion: Micro-benchmarks

We also evaluated our approach on synthetic examples of varying lengths and both 8- and 16- bit checksums to report a set of micro-benchmark performance results.

Table 6: Comparison of mean run time in seconds for enumerative and dynamic programming search strategies applied to 10 problem instances of equal message length.

8-bit Checksum	Single Index		Full Search	
Length (bytes)	Dyn.	Enum.	Dyn.	Enum.
10	0.12	0.15	0.26	0.25
50	0.49	0.55	15.54	20.77
100	1.41	2.72	125.34	231.45
150	3.24	6.76	412.61	975.29
200	5.85	13.37	1014.77	2845.92
250	8.50	24.59	1942.27	5880.57
300	12.11	37.74	3108.07	11211.78
16-bit Checksum	Single Index		Full Search	
Length (bytes)	Dyn.	Enum.	Dyn.	Enum.
10	0.11	0.14	0.14	0.13
50	0.24	0.69	2.38	2.33
100	0.48	0.58	15.78	19.91
150	0.94	1.63	54.01	84.15
200	1.49	2.67	122.27	214.53
250	2.29	4.45	240.86	490.03
300	3.39	6.71	426.80	949.37

The search space for our problem is large. Because we do not assume that the user knows the location of the checksum and the payload bounds we must search for these values over a large number of possibilities. For example, if all messages are n bytes, an 8-byte search involves searching n possible checksum locations. Additionally, we define a payload in terms of a start and end index, meaning that for each checksum location we must check $O(n^2)$ trimmed payloads.

The examples in the test suite consisted of relatively short messages ranging in length from 7 to 64 bytes. Therefore, we wanted to evaluate the performance of our approach on longer messages.

Each problem instance in our micro-benchmark consists of 5 randomly generated messages, each with a checksum calculated using the algorithm `(fold addition xs 0)` at the end of the message. We encoded value payloads by generating values randomly distributed over the byte range (0 - 255 for 8-bit sets and 0 - 65535 for 16-bit sets). We report characteristics for search restricted to only one index, simulating a search if the location of the checksum is already known and characteristics for a full search where the checksum location is unknown in Table 6.

4.3 Discussion: Corrupted Messages

Checksums are designed to detect errors, so it is reasonable to assume that messages in the data may be corrupted. Therefore, we suggest running the approach on small subsets of the messages, reducing the likelihood of noise and corruption in the data interfering in the results. A parameterization must correctly compute the checksum for every message in the set so if any message has been corrupted, the parameterization will not be found. By running our

approach on subsets of the sample messages our approach can be utilized even if some of the messages in the set have been corrupted.

5 RELATED WORK

Work related to checksum identification and inference ranges from manual methods to methods of varying degrees of automation. Automated methods include techniques for fuzzing unknown binaries as well as automatic methods of generalized protocol reverse engineering. We note that question and answer sites devoted to reverse engineering often included information about tactics and methods individuals have used when trying to solve checksum inference instances.

5.1 Ad hoc Manual Techniques

One manual technique discussed on Q&A sites relies on finding samples that differ only in a small number of bits or on a certain byte [4]. By observing the changes these differences have on the message's check bytes, an analyst may gain insight into which operation is being performed by the checksum algorithm.

If the analyst has access to the binary program used to generate messages, she can use it to probe the algorithm. Specifically, she can manufacture related samples by repeatedly incrementing a single element of program input or portion of program memory prior to the checksum calculation. She can also employ these techniques to help identify the location of the check bytes by observing what portions of the message change in relation to the altered field.

Another technique differentiates between types of error detection codes using Shannon entropy [3]. This technique leverages the observation that checksum algorithms using sum or xor operations generally produce the least amount of Shannon entropy. In contrast, CRC algorithms produce more entropy, while hashes create maximum entropy. From this knowledge, developers can gauge the most likely type of algorithm by observing the degree of randomness in the check bytes of similar messages. It is notable that subject matter experts often have insights into solving such problems that we have not found in the academic literature.

Our approach to solving the problem of checksum identification and inference does not rely on these manual methods. Instead, our approach uses differences in messages to eliminate synthesized candidate algorithms through differences between expected and produced output. While our approach does not tackle CRCs or hash functions, we do use Shannon entropy as a method identifying candidate checksum regions.

5.2 Automated and Semi-Automated Techniques

Checksum inference is generally considered to be a component of protocol reverse engineering [10, 12] or of program fuzzing [20]. We divide these approaches into two categories: first, those that assume access to a binary program that calculates the error detecting code, and second, those that assume only that they are given example messages that contain an error detecting code.

Wang *et al.* describe TaintScope [22], a directed fuzzing tool focused on automatic software vulnerability detecting. TaintScope strives to allow fuzzers to get past the gatekeeping aspect of a checksum. By correctly computing a checksum on some input, a fuzzer is able to ensure that a target program will operate on the input, rather than rejecting the majority of fuzzed cases for checksum errors. TaintScope analyzes the binary program to detect the presence of checksums and the code responsible for their calculation on file format inputs. Our approach assumes that the binary program is unavailable or would be to difficult or time consuming to instrument. Instead, our approach requires only messages captured during transmission.

Cui *et al.* introduce Tupni [2] which focuses on message format inference from both file formats and network applications. Similar to TaintScope, Tupni relies on the availability of a binary and a method to examine memory while the program is executing. Our approach does not rely on access to a running instance of the binary program.

Two works closely related to our approach are Pohl's Automatic Wireless Protocol Reverse Engineering [15] and the open source program CRC RevEng [1].

Pohl's approach focuses on general message format inference solely from network message samples assumed to originate from wireless radio protocols. Pohl's work supports the identification of message regions consistent with payload and calculated CRCs for a small set of common CRC algorithms and three wireless radio checksums. Our approach differs in two main ways. First, we focus specifically on a search space of checksums rather than specific CRCs. Second, our goal is to synthesize and return to the user runnable code for calculating a checksum rather than a message format description. Pohl's work inspired us to adopt dynamic programming to improve our performance.

CRC RevEng operates on user-specified pairs of payload and check bytes, checking 107 preset CRC algorithms as well as user-specified CRC algorithms. Our approach differs by focusing on checksums instead of CRCs. Our approach searches over the range of message indices to locate payload bounds and candidate checksum location. CRC RevEng assumes that the user has correctly provided the payload and CRC check bytes for the CRC parameter search. We note that CRC RevEng could be easily integrated into our approach. We describe this integration in the next section.

6 FUTURE WORK

With our formal model of checksum algorithms, our approach found the location and algorithm for 12 of the 14 checksum examples in our test suite. We describe several directions for future work which would expand on our results.

First, we would like to extend our model to incorporate additional operations and to allow the synthesis of more complicated checksum algorithms, such as the two instances in our testbed for which our approach was unable to produce a result.

Second, we note that in cases where our approach returns multiple solutions, an analyst needs help deciding which of these is most likely even though all are consistent with the message data. For example, an analyst may prefer a model where the payload contains as much of the original message as possible combined with checksum algorithms which are simpler. This would maximize the utility of the checksum, while minimizing the computational expense incurred. Another avenue for future work, then, is developing and improving heuristics to address this issue. We believe

application of the minimum description length principle [7] to our search results would be beneficial in this area.

While we focus on checksums in this work, it can be difficult for users to differentiate messages using checksums from those using CRCs. We would like to integrate the CRC RevEng program as a component of our search. This extension would provide a unified approach for detecting, locating and synthesizing both checksum and CRC algorithms from example messages.

At present our approach returns Python code to an analyst. Another avenue we intend to pursue is adding output for data description languages such as PADS [6].

Finally, we observe that the possibility of bit errors in messages poses a challenge to our approach. Engineers design checksums to detect errors, so it is reasonable to assume that a corrupt message may occur in a sample set. For our approach to find an algorithm, it must correctly compute the checksum for every message in the set. In the future, we would like to add an option for a lower bound for the percentage of messages that an algorithm must work for. We could then report algorithms that may not work for all messages but work for a percentage of messages above this threshold.

In summary, our approach shows promise in inferring simple checksum algorithms. Our approach produced meaningful results in the majority of cases and would significantly reduce the amount of time and effort required to manually reverse engineer the binary data.

ACKNOWLEDGMENTS

This material is based upon work partly supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No.HR0011-19-C-0073. This project was sponsored in part by the Air Force Research Laboratory (AFRL) under contract number FA8750-19-C-0039. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Air Force.

REFERENCES

[1] Gregory Cook. 2020. CRC RevEng. Retrieved June 23, 2020 from http://reveng.sourceforge.net/

[2] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. 2008. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*. 391–402.

[3] Stack Exchange. 2014. *Guessing CRC checksum algorithm.* Retrieved June 23, 2020 from https://reverseengineering.stackexchange.com/questions/4460

[4] Stack Exchange. 2014. *Reversing simple message + checksum pairs (32 bytes).* Retrieved June 23, 2020 from https://reverseengineering.stackexchange.com/questions/6927

[5] Stack Exchange. 2020. Reverse Engineering Stack Exchange. Retrieved June 23, 2020 from https://reverseengineering.stackexchange.com

[6] Kathleen Fisher and Robert Gruber. 2005. PADS: a domain-specific language for processing ad hoc data. *ACM Sigplan Notices* 40, 6 (2005), 295–304.

[7] Peter D Grünwald and Abhijit Grunwald. 2007. *The minimum description length principle.* MIT press.

[8] Sumit Gulwani. 2010. Dimensions in program synthesis. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. 13–24.

[9] Fred Halsall. 1995. *Data communications, computer networks and open systems.* Addison Wesley Longman Publishing Co., Inc. 102–112 pages.

[10] Stephan Kleber, Lisa Maile, and Frank Kargl. 2018. Survey of protocol reverse engineering algorithms: Decomposition of tools for static traffic analysis. *IEEE Communications Surveys & Tutorials* 2018 (2018).

[11] Zohar Manna and Richard Waldinger. 1980. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 2, 1 (1980), 90–121.

[12] John Narayan, Sandeep K Shukla, and T Charles Clancy. 2015. A survey of automatic protocol reverse engineering tools. *ACM Computing Surveys (CSUR)* 48, 3 (2015), 1–26.

[13] Stack Overflow. 2020. Stack Overflow. Retrieved June 23, 2020 from https://stackoverflow.com/

[14] Larry L Peterson and Bruce S Davie. 2007. *Computer networks: a systems approach.* Elsevier. 93–101 pages.

[15] Johannes Pohl and Andreas Noack. 2019. Automatic wireless protocol reverse engineering. In *13th {USENIX} Workshop on Offensive Technologies ({WOOT} 19)*.

[16] John Postel. 1981. Internet Control Message Protocol; RFC792. *ARPANET Working Group Requests for Comments* 792 (1981).

[17] Jon Postel. 1990. RFC 791: Internet Protocol, September 1981. *Darpa Internet Protocol Specification* (1990).

[18] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[19] C. E. Shannon. 1948. A mathematical theory of communication. *The Bell System Technical Journal* 27, 3 (July 1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

[20] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery.* Pearson Education.

[21] David Wagner and R Dean. 2000. Intrusion detection via static analysis. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001.* IEEE, 156–168.

[22] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 497–512.

A TUFTS CHECKSUM CORPUS EXAMPLE MESSAGES AND PARAMETERS

A.1 Example Messages

Listing 4: Example 1

```
FC620130100300001400 00BD0000000000000000000089
FC620130100300001500 00BE0000000000000000000087
FC620130100300001600 00C0000000000000000000084
FC620130100300001C00 00D000000000000000000006E
```

Listing 5: Example 2

```
AA1C010100E700000000000000002A600001429000025CAFF00000082000000
AA1C010102010000000000000002A600001429000025CAFF00000090E000000
AA1C0101095F000000000000000002A600001429000025CAFF00000003000000
```

Listing 6: Example 3

```
E1C00A4EAA6038349484271330
E1C00A1E0F249101848001B0B2
E1C00A5E2F34910186810130A5
E1C00A1F8E6431159E890333CE
```

Listing 7: Example 4

```
806FA30102B00818
806FA30112800878
1003A30001004006729E99940012120B
1003A30001003007709C98940012121F
1003A30001003806739C9B9400121202
806FA30200800041
```

Listing 8: Example 5

```
040000000000000000000000000000A9
060000000000000000000000000000AB
800024F4671EA3000000003F00FBE6F981BB17D1
880202020202020202000000000299EEA45767821
```

Table 7: Tufts Checksum Corpus Example Parameters. For example 10 two solutions were reported by forum posters.

ID	mStart	mEnd	mCheck	foldOp	finalOp	Width	Checksum Algorithm	Samples	Results
1	1	-1	21	sub	None	8	(lambda (xs) (fold subtraction xs 0))	25	104
2	0	-4	28	add	None	8	(lambda (xs) (fold addition xs 0))	252	161
3	3	-1	12	add	(add, 0x1a)	8	(lambda (xs) (addition (fold addition xs 0) 0x1a))	75	8
4	0	-1	-1	add	(xor, 0x55)	8	(lambda (xs) (xor (fold addition xs 0) 0x55))	32	2
5	0	-1	-1	add	(add, 0xa5)	8	(lambda (xs) (addition (fold addition xs 0) 0xa5))	8	2
6	2	-1	10	add	None	8	(lambda (xs) (fold addition xs 0))	25	22
7	0	-1	-1	sub	None	8	(lambda (xs) (fold subtraction xs 0))	38	64
8	0	-1	12	add	(xor, 0x55)	8	(lambda (xs) (xor (fold addition xs 0) 0x55))	12	6
9	0	0	1	Ones Comp	invert	16	(lambda (xs) (invert (fold onesComplement xs 0)))	7	20
10 (a)	0	-2	14	add	None	16	(lambda (xs) (fold addition xs 0))	14	13
10 (b)	14	-1	15	sub	(add,0xFFF2)	16	(lambda (xs) (addition (fold subtraction xs 0) 0xFFF2))	-	-
11 (IPv4 Header)	0	0	5	Ones Comp	Invert	16	(lambda (xs) (invert (fold onesComplement xs 0)))	7	24
12 (ICMP)	0	0	1	Ones Comp	Invert	16	(lambda (xs) (invert (fold onesComplement xs 0)))	10	48

Listing 9: Example 6

```
55549D1748C2AA148102FF
55549D1748C2AA1D8E0417
55549D1748C2AA286502F7
```

Listing 10: Example 7

```
00010302D20226
2001030A4100000024A0030895BA73
20010312D1402001000021010000220100002301000002F
20010302D240C8
```

Listing 11: Example 8

```
A802000000001E5D0000000070
A80300000004055D0000000044
A81200004004159A00000000F8
A85200084000149C00000000A7
```

Listing 12: Example 9

```
D00771BCBE3B0000
0B00396CBE3B05007E5246436172644F6E
D0073532BE3B07007E4465766963654E616D653D544138313000
D0074ED4E1230000
```

Listing 13: Example 10

```
FFFFFFFF01671F840000000000000000000000000000000000101002
1EADE08
3065C0000000000000000000000000000000000000006F85000101006
0EB9F07
FFFFFFFF01671F8C0000000000000000000000000000000000101002
1F2DE00
```

Listing 14: Example 11

```
4500002894D00000710654CC0D6B8809C0A80A17
450000282BA700007106BDF50D6B8809C0A80A17
45000028BD27000071062C750D6B8809C0A80A17
4500012C000040004006D998C0A80A170D6B8809
450000520000400040006DA72C0A80A170D6B8809
```

Listing 15: Example 12

```
08009654A41700005EAB245C00044F8508090A0B0C0D0E0F101112131
415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2F30
31323334353637
00009E54A41700005EAB245C00044F8508090A0B0C0D0E0F101112131
415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2F30
31323334353637
08008239A41700015EAB245D0004639E08090A0B0C0D0E0F101112131
415161718191A1B1C1D1E1F202122232425262728292A2B2C2D2E2F30
31323334353637
```