Static Compilation of Regular Expressions for Analysis and Modification

Rob King

DVLabs
TippingPoint Technologies

February 18, 2010

OUTLINE

THE PROBLEM

In which we discover purpose of the whole thing...

- This talk is about inspecting streams of data for interesting patterns, even when that stream of data has been encoded.
- We focus on the Base64 encoding scheme, and discuss a tool that can be used when dealing with Base64.
- However, most portions of the algorithm are applicable to other position-dependent bitwise block encodings (and, potentially, self-synchronizing encodings).

THE PROBLEM

In which we discover purpose of the whole thing...

- This talk is about inspecting streams of data for interesting patterns, even when that stream of data has been encoded.
- We focus on the Base64 encoding scheme, and discuss a tool that can be used when dealing with Base64.
- However, most portions of the algorithm are applicable to other position-dependent bitwise block encodings (and, potentially, self-synchronizing encodings).

THE PROBLEM

In which we discover purpose of the whole thing...

- This talk is about inspecting streams of data for interesting patterns, even when that stream of data has been encoded.
- We focus on the Base64 encoding scheme, and discuss a tool that can be used when dealing with Base64.
- However, most portions of the algorithm are applicable to other position-dependent bitwise block encodings (and, potentially, self-synchronizing encodings).

THE PROBLEM

In which we discover purpose of the whole thing...

- This talk is about inspecting streams of data for interesting patterns, even when that stream of data has been encoded.
- We focus on the Base64 encoding scheme, and discuss a tool that can be used when dealing with Base64.
- However, most portions of the algorithm are applicable to other position-dependent bitwise block encodings (and, potentially, self-synchronizing encodings).

CHALLENGES OF DATA STREAM INSPECTION

When looking for patterns in streams of data, several things must be kept in mind:

- There is no "luxury of time".
- Context is limited.
- Resources are limited.

CHALLENGES OF DATA STREAM INSPECTION

When looking for patterns in streams of data, several things must be kept in mind:

- There is no “luxury of time”.
- Context is limited.
- Resources are limited.

CHALLENGES OF DATA STREAM INSPECTION

When looking for patterns in streams of data, several things must be kept in mind:

- There is no “luxury of time”.
- Context is limited.
- Resources are limited.

ENCODED STREAMS

- Sometimes, the streams we're inspecting will be encoded.
- This means that we're going to have to be (more!) clever when looking for patterns in these streams.

DEALING WITH ENCODED STREAMS

There are several general strategies for dealing with encoded streams.

DEALING WITH ENCODED STREAMS

Strategy 1: Ignore the Encoding

- The easiest thing to do is simply pretend the stream is not encoded at all.
- Advantages:
  - We're already done.
- Disadvantages:
  - We're essentially admitting defeat.
  - Whatever we were looking for is not going to be found.
  - We're stil burdening our analysis engine with lots of data with which we can do nothing.

- The easiest thing to do is simply pretend the stream is not encoded at all.
- Advantages:
  - We're already done.
- Disadvantages:
  - We're essentially admitting defeat.
  - Whatever we were looking for is not going to be found.
  - We're stil burdening our analysis engine with lots of data with which we can do nothing.

DEALING WITH ENCODED STREAMS

Strategy 1: Ignore the Encoding

- The easiest thing to do is simply pretend the stream is not encoded at all.
- Advantages:
  - We're already done.
- Disadvantages:
  - We're essentially admitting defeat.
  - Whatever we were looking for is not going to be found.
  - We're stil burdening our analysis engine with lots of data with which we can do nothing.

- Rather than pretending that the data is not encoded, we could go one step further, and detect that the data is encoded.
- Once we've detected that the data is encoded, we can simply drop the stream.
- Advantages:
    - Stops burdening with inspection engine with data we know we can't inspect.
- Disadvantages:
    - If we "fail open" and allow encoded data to pass without inspection, we just gave anyone who wants to bypass our inspection a "get out of jail free" card.
    - If we "fail closed" and block encoded data, we just blocked all legitimate uses of that encoding scheme.

- Rather than pretending that the data is not encoded, we could go one step further, and detect that the data is encoded.
- Once we've detected that the data is encoded, we can simply drop the stream.
- Advantages:
    - Stops burdening with inspection engine with data we know we can't inspect.
- Disadvantages:
    - If we "fail open" and allow encoded data to pass without inspection, we just gave anyone who wants to bypass our inspection a "get out of jail free" card.
    - If we "fail closed" and block encoded data, we just blocked all legitimate uses of that encoding scheme.

DEALING WITH ENCODED STREAMS

Strategy 2: Ignore the Data

- Rather than pretending that the data is not encoded, we could go one step further, and detect that the data is encoded.
- Once we've detected that the data is encoded, we can simply drop the stream.
- Advantages:
  - Stops burdening with inspection engine with data we know we can't inspect.
- Disadvantages:
  - If we "fail open" and allow encoded data to pass without inspection, we just gave anyone who wants to bypass our inspection a "get out of jail free" card.
  - If we "fail closed" and block encoded data, we just blocked all legitimate uses of that encoding scheme.

DEALING WITH ENCODED STREAMS

Strategy 2: Ignore the Data

- Rather than pretending that the data is not encoded, we could go one step further, and detect that the data is encoded.
- Once we've detected that the data is encoded, we can simply drop the stream.
- Advantages:
  - Stops burdening with inspection engine with data we know we can't inspect.
- Disadvantages:
  - If we "fail open" and allow encoded data to pass without inspection, we just gave anyone who wants to bypass our inspection a "get out of jail free" card.
  - If we "fail closed" and block encoded data, we just blocked all legitimate uses of that encoding scheme.

- We could buffer the entirety of an encoded stream.
- Once we've buffered the whole stream, we can decode it, inspect it, reencode it, and send it on its way.
- Advantages:
  - We get the complete power of our inspection engine.
- Disadvantages:
  - Latency becomes unbounded.
  - Resource usage becomes unbounded.
  - We have to modify the engine for every encoding we need to inspect.
- These advantages and disadvantages apply roughly equally to any streaming decoder.

- We could buffer the entirety of an encoded stream.
- Once we've buffered the whole stream, we can decode it, inspect it, reencode it, and send it on its way.
- Advantages:
  - We get the complete power of our inspection engine.
- Disadvantages:
  - Latency becomes unbounded.
  - Resource usage becomes unbounded.
  - We have to modify the engine for every encoding we need to inspect.
- These advantages and disadvantages apply roughly equally to any streaming decoder.

- We could buffer the entirety of an encoded stream.
- Once we've buffered the whole stream, we can decode it, inspect it, reencode it, and send it on its way.
- Advantages:
  - We get the complete power of our inspection engine.
- Disadvantages:
  - Latency becomes unbounded.
  - Resource usage becomes unbounded.
  - We have to modify the engine for every encoding we need to inspect.
- These advantages and disadvantages apply roughly equally to any streaming decoder.

- We could buffer the entirety of an encoded stream.
- Once we've buffered the whole stream, we can decode it, inspect it, reencode it, and send it on its way.
- Advantages:
  - We get the complete power of our inspection engine.
- Disadvantages:
  - Latency becomes unbounded.
  - Resource usage becomes unbounded.
  - We have to modify the engine for every encoding we need to inspect.
- These advantages and disadvantages apply roughly equally to any streaming decoder.

- We could buffer the entirety of an encoded stream.
- Once we've buffered the whole stream, we can decode it, inspect it, reencode it, and send it on its way.
- Advantages:
  - We get the complete power of our inspection engine.
- Disadvantages:
  - Latency becomes unbounded.
  - Resource usage becomes unbounded.
  - We have to modify the engine for every encoding we need to inspect.
- These advantages and disadvantages apply roughly equally to any streaming decoder.

- Instead of decoding the data, we could encode the pattern.
- Advantages:
  - We get (most?) of the power of our inspection engine.
  - There is no performance penalty for decoding.
  - Resource usage and latency are bounded.
  - We need not buffer or store context.
- Disadvantages:
  - A different transform must be written for each scheme.
  - Might not be possible for some schemes or patterns.
  - False positives may become more common.

DEALING WITH ENCODED STREAMS

Strategy 4: Modify the Pattern

- Instead of decoding the data, we could encode the pattern.
- Advantages:
  - We get (most?) of the power of our inspection engine.
  - There is no performance penalty for decoding.
  - Resource usage and latency are bounded.
  - We need not buffer or store context.
- Disadvantages:
  - A different transform must be written for each scheme.
  - Might not be possible for some schemes or patterns.
  - False positives may become more common.

- Instead of decoding the data, we could encode the pattern.
- Advantages:
    - We get (most?) of the power of our inspection engine.
    - There is no performance penalty for decoding.
    - Resource usage and latency are bounded.
    - We need not buffer or store context.
- Disadvantages:
    - A different transform must be written for each scheme.
    - Might not be possible for some schemes or patterns.
    - False positives may become more common.

- Base64 is an Internet standard for encoding arbitrary (usually binary) data using only printable characters common in many character sets.
- Multiple minor variants, but the most common is defined in RFC4648.
- Base64 is used in numerous situations, essentially whenever binary data needs to be encoded in a printable form.

The most common application of Base64 is in the encoding of attachments to email messages.

```
--Apple-Mail-35-294643828
Content-Disposition: inline;
	filename=stage1.png
Content-Type: image/png;
	x-mac-hide-extension=yes;
	x-unix-mode=0644;
	name="stage1.png"
Content-Transfer-Encoding: base64
```


```
iVBORw0KGgoAAAANSUhEUgAAAGAAAAIbCAYAAAAU4psTAAAACXBIWXMAAAsTAAALEwEAmpwYAAAg
AElEQVR4Ae2dCfQUxfHH2yTGnF6JRqNoJGIkiQbBG/ECFFBBUbzQiKJ4PA4PRLyiEN7zofEp4gFR
8QQUFPEEDD5AoKhFQPEnAmEMTJRdiYhJjYv/rU/nPvv3tb3d2eo7tnd9v6r3d2Z3p6a761nRPH9VV
61khU5A3BD7jreSiYEWgUIDnB+FznsuPXPxHH31k3nvvPfOXv/zF/POf/zT/+te/9EgGX/ziF80X
vvAFPX7ta18zW265pfnyl78cOW+fCZtKAbyO3nzzzTbNixQrxhtv609f/vKX5ve//73529/+5oTT
V7/6VbPVluZ73zn0+a73/2u+d73vme6d0miv9dbbz2nvLJMvJ7vlzBAP/bYY+bZZ581ixcvNmvX
rjWf//znzQ477KBgde7c2Wy99db6VPNkb7bZZqUnnqceojYENeJPf/qT1hRqy7vvvvmtWrlypily1
apX597//bTbZZBzzz77mB49eph+/fqpYrIEuF7eXhTw0ksvzZFz5q5ZnNv95c+eaet956y2y++eZmv/32
U2AAZ+eddzaf+1y6fM///mPefXVV1XJKPpnP/uZ+eMf/2i23357c/jhh5vjjz/e7LLLLvXwSv86
NaAR90GHH9qbbbrrJdu3alW6v7dSpkx0zZox97rn7H//+99GsNCiDMqbbHiAF3iCN3iE10aRybbqg
v/71r/ayyy6zUvXtl77JXvSSSdZaW6yyzyeCbrrfWOBDrJUABzHnbpDt45P7tEbPxrECqG/7BwDnGU
MnStuQsP7yOEOwmhRbxh7LHfmyIlJYP0PkvlA9SDBMsvUQZpvUMovJTAonAN2/MqKTvWzDWeVRu6
8M+oKw1J8TxGnY+kYRo9KnMNpBbxN/mFUTe2sN4xIHd8Ea0l42WMDTG2nBuP7PrNXVPQzSExFj+5GaM
VsW9eVEvzZCWjT+/vqq71LgaS2aDU6bu6tBZlGtiClzVU1Wd7V/QCw8sNkuHKDQTKXM1qOpx6fW+ktgn
hB8W3/rxR9OMQ0oJKs/3QdZVkpKo07vMl1ajjFhxkfUdMn2PSkvm0k4BWCB3agcR/bOrnT5P03Wa5FOQ/
Hy7BNQ8h3cMw9bZvrl3ap8N4HJBYQ3wyA2UwcWLNFbJHsVpnkSxkdT35Uou3WF2uDJQWdhvMgfgDsqW
5Uro3qq1FNvEsd9o9mfr6w3+6bYE9udNjSKbIDAQCJSqDXXM6zJ21FLP1nmTK7wRFzsVdRHM3tDYE6
4f0a1U0cFDQ1pdFe5ejM5vmN3f2A+5MXNRRAjXNYHv8UCJ00hjRCjDMw/bqGpcxkWVn1eVjJ9+Tk95u
```

- Base64 expects input as a series of eight-bit octets.
- Every three octets are grouped together into a collection of 24-bits.
- These 24-bits are then split into four six-bit sextets.
- Each sextet is used as a big-endian index into the zero-based array that is the Base64 alphabet.

f								o								o							
0	1	1	0	0	1	1	0	0	1	1	0	1	1	1	1	0	1	1	0	1	1	1	1
Z						m						9						v					

- If the input is not evenly divisible by three, it is right-padded by one or two octets of zeroes, and one or two “=” symbols are appended to the output.

BASE64

Base64 Input

- Base64 expects input as a series of eight-bit octets.
- Every three octets are grouped together into a collection of 24-bits.
- These 24-bits are then split into four six-bit sextets.
- Each sextet is used as a big-endian index into the zero-based array that is the Base64 alphabet.

<table>
<tr><td colspan="8">f</td><td colspan="8">o</td><td colspan="8">o</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="6">Z</td><td colspan="6">m</td><td colspan="6">9</td><td colspan="6">v</td></tr>
</table>

- If the input is not evenly divisible by three, it is right-padded by one or two octets of zeroes, and one or two “=” symbols are appended to the output.

BASE64

Base64 Input

- Base64 expects input as a series of eight-bit octets.
- Every three octets are grouped together into a collection of 24-bits.
- These 24-bits are then split into four six-bit sextets.
- Each sextet is used as a big-endian index into the zero-based array that is the Base64 alphabet.

<table>
<tr><td colspan="8">f</td><td colspan="8">o</td><td colspan="8">o</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="6">Z</td><td colspan="6">m</td><td colspan="6">9</td><td colspan="6">v</td></tr>
</table>

- If the input is not evenly divisible by three, it is right-padded by one or two octets of zeroes, and one or two “=” symbols are appended to the output.

BASE64
Base64 Input

- Base64 expects input as a series of eight-bit octets.
- Every three octets are grouped together into a collection of 24-bits.
- These 24-bits are then split into four six-bit sextets.
- Each sextet is used as a big-endian index into the zero-based array that is the Base64 alphabet.

<table>
<tr><td colspan="8">f</td><td colspan="8">o</td><td colspan="8">o</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr>
<tr><td colspan="6">Z</td><td colspan="6">m</td><td colspan="6">9</td><td colspan="6">v</td></tr>
</table>

- If the input is not evenly divisible by three, it is right-padded by one or two octets of zeroes, and one or two “=” symbols are appended to the output.

- Base64 expects input as a series of eight-bit octets.
- Every three octets are grouped together into a collection of 24-bits.
- These 24-bits are then split into four six-bit sextets.
- Each sextet is used as a big-endian index into the zero-based array that is the Base64 alphabet.

f								o								o							
0	1	1	0	0	1	1	0	0	1	1	0	1	1	1	1	0	1	1	0	1	1	1	1
Z						m						9						v					

- If the input is not evenly divisible by three, it is right-padded by one or two octets of zeroes, and one or two "=" symbols are appended to the output.

- Base64 expects input as a series of eight-bit octets.
- Every three octets are grouped together into a collection of 24-bits.
- These 24-bits are then split into four six-bit sextets.
- Each sextet is used as a big-endian index into the zero-based array that is the Base64 alphabet.

f								o								o							
0	1	1	0	0	1	1	0	0	1	1	0	1	1	1	1	0	1	1	0	1	1	1	1
Z						m						9						v					

- If the input is not evenly divisible by three, it is right-padded by one or two octets of zeroes, and one or two "=" symbols are appended to the output.

BASE64
The Base64 Alphabet

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e	f
2	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v
3	w	x	y	z	0	1	2	3	4	5	6	7	8	9	+	/
The “=” sign is used for padding.																

- Base64 is position dependent.
- In general, any given string will be encoded in one of three different ways, depending on its offset into the input.
- This is what makes the encoding of patterns so hard.

Encoding of “foo” at Three Different Offsets	
Offset 0	Zm9v
Offset 1	[159BFJNRVZdhlptx]mb2[+/8-9]
Offset 2	[2GWm]Zvb[+/0-9w-z]

REGULAR EXPRESSIONS

Choosing a Pattern Language

- Our choice of pattern language controls the overall complexity of patterns for which we can search.
- If we were just looking for static strings, we wouldn't need a new tool - just use grep.
- Regular expressions provide a good balance of ease of implementation, expressive power, and common availability.

- Formalized by Stephen Kleene in 1956.
- Ken Thompson incorporated them as a useful pattern matching tool into his version of the QED editor for MIT's CTSS timesharing system.
- This later influenced Ken Thompson's implementation of `ed` for UNIX.
- From UNIX, regular expressions spread around the world.

SUPPORTED REGULAR EXPRESSION OPERATIONS

- Character matches (e.g. “a”)
- Concatenation (e.g. “ab”)
- Alternation (e.g. “(ab|cd)”)
- Character classes and inverse classes (e.g. “[0 – 9]”)
- Kleene closures (e.g. “A*C”)

UNSUPPORTED REGULAR EXPRESSION OPERATIONS

- Backreferences and captures
- Variable-length repetition
- Left and right anchors
- Just about everything else

RESTATEMENT OF PURPOSE

- Come up with a way to transform an arbitrary regular expression such that it will match its input when that input has been encoded using the Base64 algorithm.
- Do this transformation in such a way that the regular expression does not grow too large.
- Do this transformation in such a way that not too much information is lost.
- Do this transformation in such a way that the expression will match regardless of the pattern's offset from the beginning of input.

RESTATEMENT OF PURPOSE

- Come up with a way to transform an arbitrary regular expression such that it will match its input when that input has been encoded using the Base64 algorithm.
- Do this transformation in such a way that the regular expression does not grow too large.
- Do this transformation in such a way that not too much information is lost.
- Do this transformation in such a way that the expression will match regardless of the pattern's offset from the beginning of input.

RESTATEMENT OF PURPOSE

- Come up with a way to transform an arbitrary regular expression such that it will match its input when that input has been encoded using the Base64 algorithm.
- Do this transformation in such a way that the regular expression does not grow too large.
- Do this transformation in such a way that not too much information is lost.
- Do this transformation in such a way that the expression will match regardless of the pattern's offset from the beginning of input.

RESTATEMENT OF PURPOSE

- Come up with a way to transform an arbitrary regular expression such that it will match its input when that input has been encoded using the Base64 algorithm.
- Do this transformation in such a way that the regular expression does not grow too large.
- Do this transformation in such a way that not too much information is lost.
- Do this transformation in such a way that the expression will match regardless of the pattern's offset from the beginning of input.

- Convert the regular expression into a nondeterministic finite state automaton using Thompson's algorithm, then convert the NFA to a deterministic finite state automaton using the powerset construction algorithm, then transform the DFA into a directed acyclic graph, then perform graph reductions until the graph is in a minimal form, transform the minimal form using graph transformations, then reduce again, then serialize the graph as a regular expression.
- This is hard, and I'm lazy.

WAYS OF SOLVING THE PROBLEM

The Right Way

- Convert the regular expression into a nondeterministic finite state automaton using Thompson's algorithm, then convert the NFA to a deterministic finite state automaton using the powerset construction algorithm, then transform the DFA into a directed acyclic graph, then perform graph reductions until the graph is in a minimal form, transform the minimal form using graph transformations, then reduce again, then serialize the graph as a regular expression.
- This is hard, and I'm lazy.

WAYS OF SOLVING THE PROBLEM

The Right Way

- Convert the regular expression into a nondeterministic finite state automaton using Thompson's algorithm, then convert the NFA to a deterministic finite state automaton using the powerset construction algorithm, then transform the DFA into a directed acyclic graph, then perform graph reductions until the graph is in a minimal form, transform the minimal form using graph transformations, then reduce again, then serialize the graph as a regular expression.
- This is hard, and I'm lazy.

WAYS OF SOLVING THE PROBLEM

The Wrong Way

- Enumerate all possible matching strings for a regular expression.
- Encode these strings and concatenate them inside an alternating regular expression.
- This would definitely work, with one problem...
- Regular expressions with Kleene closures can match an infinite number of strings.
- Enumerating an infinite number of strings can take a really long time.
- Even the trivial regular expression "...." would match over four billion strings.

- Enumerate all possible matching strings for a regular expression.
- Encode these strings and concatenate them inside an alternating regular expression.
- This would definitely work, with one problem...
- Regular expressions with Kleene closures can match an infinite number of strings.
- Enumerating an infinite number of strings can take a really long time.
- Even the trivial regular expression "...." would match over four billion strings.

WAYS OF SOLVING THE PROBLEM

The Wrong Way

- Enumerate all possible matching strings for a regular expression.
- Encode these strings and concatenate them inside an alternating regular expression.
- This would definitely work, with one problem...
- Regular expressions with Kleene closures can match an infinite number of strings.
- Enumerating an infinite number of strings can take a really long time.
- Even the trivial regular expression "...." would match over four billion strings.

WAYS OF SOLVING THE PROBLEM

The Wrong Way

- Enumerate all possible matching strings for a regular expression.
- Encode these strings and concatenate them inside an alternating regular expression.
- This would definitely work, with one problem...
- Regular expressions with Kleene closures can match an infinite number of strings.
- Enumerating an infinite number of strings can take a really long time.
- Even the trivial regular expression "...." would match over four billion strings.

WAYS OF SOLVING THE PROBLEM

The Wrong Way

- Enumerate all possible matching strings for a regular expression.
- Encode these strings and concatenate them inside an alternating regular expression.
- This would definitely work, with one problem...
- Regular expressions with Kleene closures can match an infinite number of strings.
- Enumerating an infinite number of strings can take a really long time.
- Even the trivial regular expression "...." would match over four billion strings.

WAYS OF SOLVING THE PROBLEM
The Wrong Way

- Enumerate all possible matching strings for a regular expression.
- Encode these strings and concatenate them inside an alternating regular expression.
- This would definitely work, with one problem...
- Regular expressions with Kleene closures can match an infinite number of strings.
- Enumerating an infinite number of strings can take a really long time.
- Even the trivial regular expression "...." would match over four billion strings.

WAYS OF SOLVING THE PROBLEM

The Wrong Way

- Enumerate all possible matching strings for a regular expression.
- Encode these strings and concatenate them inside an alternating regular expression.
- This would definitely work, with one problem...
- Regular expressions with Kleene closures can match an infinite number of strings.
- Enumerating an infinite number of strings can take a really long time.
- Even the trivial regular expression "...." would match over four billion strings.

WAYS OF SOLVING THE PROBLEM

The Wrong Way, But Faster!

- Enumerating all possible matching strings would, in theory, give us the correct answer.
- Instead of enumerating all possible strings and encoding them, what if we encoded the operations?
- In other words, what if we enumerated only what could match a given operation at a given point in time?

WAYS OF SOLVING THE PROBLEM

The Wrong Way, But Faster!

- Enumerating all possible matching strings would, in theory, give us the correct answer.
- Instead of enumerating all possible strings and encoding them, what if we encoded the operations?
- In other words, what if we enumerated only what could match a given operation at a given point in time?

WAYS OF SOLVING THE PROBLEM

The Wrong Way, But Faster!

- Enumerating all possible matching strings would, in theory, give us the correct answer.
- Instead of enumerating all possible strings and encoding them, what if we encoded the operations?
- In other words, what if we enumerated only what could match a given operation at a given point in time?

WAYS OF SOLVING THE PROBLEM

The Wrong Way, But Faster!

- Enumerating all possible matching strings would, in theory, give us the correct answer.
- Instead of enumerating all possible strings and encoding them, what if we encoded the operations?
- In other words, what if we enumerated only what could match a given operation at a given point in time?

THE ALGORITHM

In a nutshell...

- Create a list of regular expressions that match a fixed-length string.
- Enumerate the bitstrings matched by each of these expressions.
- Break each of these bitstrings into six-bit units.
- Encode each of these units.
- Treat each of these encoded strings as a branch in an *n*-way alternation in a regular expression.
- Optimize the expression.
- There is special handling for the two “meta” operations: alternations and closures.

- Create a list of regular expressions that match a fixed-length string.
- Enumerate the bitstrings matched by each of these expressions.
- Break each of these bitstrings into six-bit units.
- Encode each of these units.
- Treat each of these encoded strings as a branch in an *n*-way alternation in a regular expression.
- Optimize the expression.
- There is special handling for the two “meta” operations: alternations and closures.

- Create a list of regular expressions that match a fixed-length string.
- Enumerate the bitstrings matched by each of these expressions.
- Break each of these bitstrings into six-bit units.
- Encode each of these units.
- Treat each of these encoded strings as a branch in an *n*-way alternation in a regular expression.
- Optimize the expression.
- There is special handling for the two “meta” operations: alternations and closures.

THE ALGORITHM

In a nutshell...

- Create a list of regular expressions that match a fixed-length string.
- Enumerate the bitstrings matched by each of these expressions.
- Break each of these bitstrings into six-bit units.
- Encode each of these units.
- Treat each of these encoded strings as a branch in an *n*-way alternation in a regular expression.
- Optimize the expression.
- There is special handling for the two “meta” operations: alternations and closures.

THE ALGORITHM

In a nutshell...

- Create a list of regular expressions that match a fixed-length string.
- Enumerate the bitstrings matched by each of these expressions.
- Break each of these bitstrings into six-bit units.
- Encode each of these units.
- Treat each of these encoded strings as a branch in an *n*-way alternation in a regular expression.
- Optimize the expression.
- There is special handling for the two “meta” operations: alternations and closures.

THE ALGORITHM

In a nutshell...

- Create a list of regular expressions that match a fixed-length string.
- Enumerate the bitstrings matched by each of these expressions.
- Break each of these bitstrings into six-bit units.
- Encode each of these units.
- Treat each of these encoded strings as a branch in an *n*-way alternation in a regular expression.
- Optimize the expression.
- There is special handling for the two “meta” operations: alternations and closures.

THE ALGORITHM

In a nutshell...

- Create a list of regular expressions that match a fixed-length string.
- Enumerate the bitstrings matched by each of these expressions.
- Break each of these bitstrings into six-bit units.
- Encode each of these units.
- Treat each of these encoded strings as a branch in an *n*-way alternation in a regular expression.
- Optimize the expression.
- There is special handling for the two “meta” operations: alternations and closures.

THE ALGORITHM

In a nutshell...

- Create a list of regular expressions that match a fixed-length string.
- Enumerate the bitstrings matched by each of these expressions.
- Break each of these bitstrings into six-bit units.
- Encode each of these units.
- Treat each of these encoded strings as a branch in an *n*-way alternation in a regular expression.
- Optimize the expression.
- There is special handling for the two “meta” operations: alternations and closures.

- Non-fixed-length operations leave ambiguity as to what bits to place in a given six-bit unit.
- The goal of this phase of the algorithm is to get a collection of regular expressions that match fixed-length strings.
- First, let's go over how to encode the atomic operations.

ENCODING OPERATIONS
Single Characters

This is a fairly obvious - a single character is encoded as the bitstring for that character.

f							
0	1	1	0	0	1	1	0

Character classes are stored as a list of all characters included in the class. This is not the most efficient possible choice in terms of memory usage, but its implementation is simpler and it is often faster.

[ABC]							
0	1	0	0	0	0	0	1
0	1	0	0	0	0	1	0
0	1	0	0	0	0	1	1

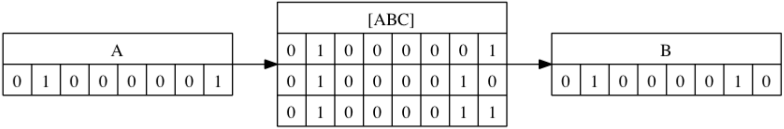
Inverted character classes simply store the complement of the list of indicated characters.

Wildcards are simply encoded as a character class with 256 entries. In the underlying implementation they are encoded as a special atom to save space, but from the point of view of the algorithm there is no difference.

ENCODING OPERATIONS

Concatenation

- Single characters, character classes, and wildcards are treated as the building blocks of lists.
- Each fixed-length expression is simply a list of these operations.
- Therefore, concatenation is modelled implicitly by the ordering of these operations in the list.



- It was stated earlier that the algorithm deals only with regular expressions that match only fixed-length strings.
- Alternations can easily violate this: if one branch of an alternation has more character matches than the other, the expression overall is not fixed length.

ENCODING ALTERNATIONS

Alternations and Fixed Lengths

- It was stated earlier that the algorithm deals only with regular expressions that match only fixed-length strings.
- Alternations can easily violate this: if one branch of an alternation has more character matches than the other, the expression overall is not fixed length.

ENCODING ALTERNATIONS

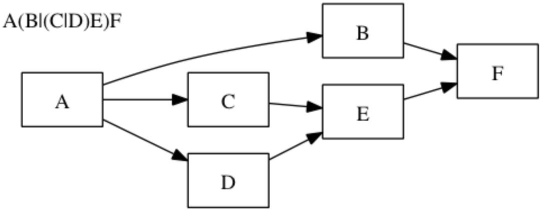
Alternations and Fixed Lengths

- It was stated earlier that the algorithm deals only with regular expressions that match only fixed-length strings.
- Alternations can easily violate this: if one branch of an alternation has more character matches than the other, the expression overall is not fixed length.

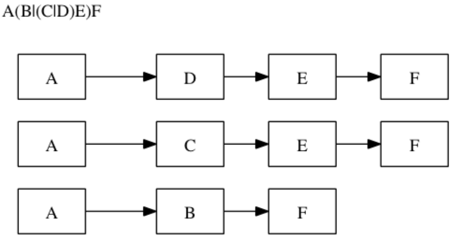
ENCODING ALTERNATIONS

Alternations as Directed Acyclic Graphs

- Alternations turn our simple lists of operations into directed acyclic graphs.
- For example, below is the representation of the regular expression “A(B|(C|D)E)F”.



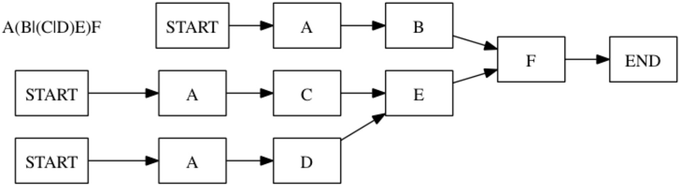
- We can easily create a list of fixed-length expressions by enumerating all topological sorts of the graph.
- For each distinct topological sort, we can create a list of fixed-length operations.
- This reduces an alternation to a fixed-length structure, and can be performed recursively, resulting in a list of expressions, each of a fixed-length.



ENCODING ALTERNATIONS

The Mythical End

- By adding an explicit "end of expression" pseudo-operation to the end of every expression, we can guarantee that all expressions have a shared tail.
- We can also add some number of explicit "start of expression" pseudo-operations from every entry point into the expression.
- These operations does not actually match anything; they exist solely for ease of implementation.



ENCODING ALTERNATIONS

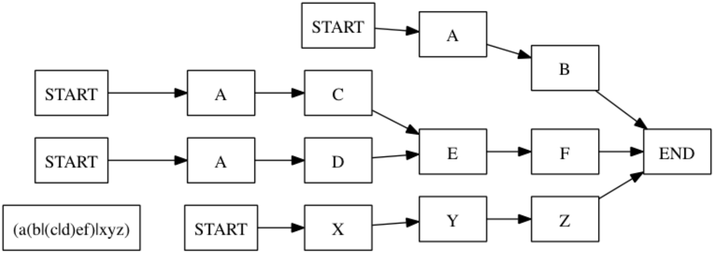
Encoding Each Fixed-Length Expression

- By finding a path from every START node to the END node, we can produce a list of fixed-length expressions.
- Note that this is essentially equivalent of the list of topological sorts, but provides the storage advantage of a shared tail.
- Implementation is also considerably easier and performs much better, since we don't need to implement a general topological sort; rather we can simply maintain a list of all start nodes and follow the only eminating edge from every node until we reach the end.

ENCODING ALTERNATIONS

Recursive Application

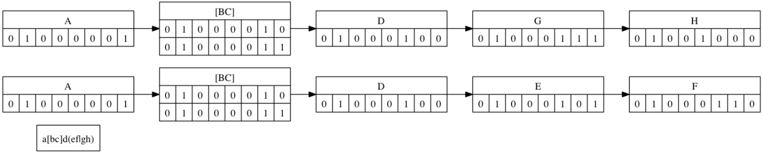
- We can apply this recursively, to build up a list of fixed-length expressions from a regular expression, even with nested alternations.
- Each of these fixed length expressions can be encoded easily using the methods for the fixed-length operations above.



ENCODING EXPRESSIONS WITHOUT CLOSURES

Create a List of Fixed Length Bit Expressions

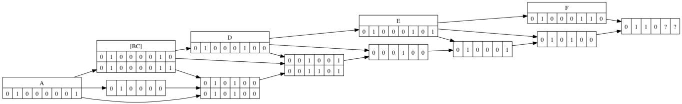
- Using the algorithm above, we can create a list of fixed-length expressions consisting of eight bit units (for single characters) or lists of eight bit units (for character classes and wildcards).



ENCODING EXPRESSIONS WITHOUT CLOSURES

Extract Six Bits at a Time

- We now use a consumer function to walk through each fixed-length expression, extracting six bits at a time.
- This produces a new list of six-bit units.
- When the consumer function must get bits from a single character and a character class, or two classes, a list of six bit units is placed in the result list.



- Since the expression is going to be examining data in a streaming context, left and right anchors don't make sense in most situations.
- Therefore, if any bits are needed to complete a six-bit unit at the beginning or the end of the expression, they are treated as if a wildcard were present immediately before or after the expression.

ENCODING EXPRESSIONS WITHOUT CLOSURES

The Initial and Final Bits

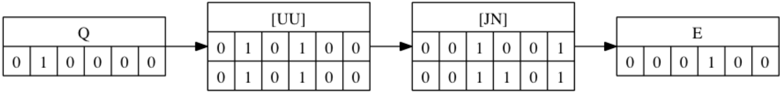
- Since the expression is going to be examining data in a streaming context, left and right anchors don't make sense in most situations.
- Therefore, if any bits are needed to complete a six-bit unit at the beginning or the end of the expression, they are treated as if a wildcard were present immediately before or after the expression.

ENCODING EXPRESSIONS WITHOUT CLOSURES

The Initial and Final Bits

- Since the expression is going to be examining data in a streaming context, left and right anchors don't make sense in most situations.
- Therefore, if any bits are needed to complete a six-bit unit at the beginning or the end of the expression, they are treated as if a wildcard were present immediately before or after the expression.

- Each six bit unit is then replaced with its equivalent symbol in the Base64 alphabet.
- For lists of six bit units (from character classes), a list of Base64 symbols is produced.



ENCODING EXPRESSIONS WITHOUT CLOSURES

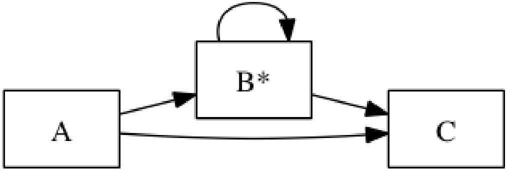
The Final Expression

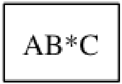
- The list of fixed-length encoded expressions is joined into an *n*-way alternation, which is treated as a regular expression.
- This expression is first run through the optimizer, which recursively refactors common prefixes and suffixes.
- The expression is then reprocessed two more times, to account for varying offsets from the beginning of input.

ENCODING EXPRESSIONS WITH CLOSURES

Violating the Fixed Length and Acyclic Rules

- Expressions with Kleene closures violate the fixed-length rule, since subexpressions can appear from zero to infinitely many times.
- Expressions with Kleene closures violate the acylic rule, since repeating expressions would become cycles in the graph representation.





ENCODING EXPRESSIONS WITH CLOSURES

The Easy Case

- The easiest case for encoding closures is the case where they don't appear at all.
- Whenever a closure is present in an expression, it acts as a virtual alternation, with one branch having at least one instance of the closure, and the other branch having no instances.

- Closures are encoded three times, to account for varying offsets from the beginning of input.
- These three encoded expressions are placed in a special data structure that marks them as the components of a closure.

ENCODING EXPRESSIONS WITH CLOSURES

A List of Expressions

- Expressions with closures are treated as a list of expressions.
- The expressions are split such that each specially-encoded closure is a separate expression.
- Lists of expressions are encoded such that the implicit wildcard at the beginning and ending of each expression is replaced with the appropriate values from the next or previous expression in the list.

EXPRESSION OPTIMIZATION

Common Prefix and Suffix Refactoring

- The expression is treated as a list of constant-length expressions, from each possible starting expression to the logical end.
- All of these expressions will differ only around places where alternations were present.
- Therefore, everything leading up to the alternation, and everything after the alternation, will be identical for various expressions.
- We can therefore optimize the expression by refactoring all common prefixes and suffixes.

- The expression is treated as a list of constant-length expressions, from each possible starting expression to the logical end.
- All of these expressions will differ only around places where alternations were present.
- Therefore, everything leading up to the alternation, and everything after the alternation, will be identical for various expressions.
- We can therefore optimize the expression by refactoring all common prefixes and suffixes.

EXPRESSION OPTIMIZATION

Common Prefix and Suffix Refactoring

- The expression is treated as a list of constant-length expressions, from each possible starting expression to the logical end.
- All of these expressions will differ only around places where alternations were present.
- Therefore, everything leading up to the alternation, and everything after the alternation, will be identical for various expressions.
- We can therefore optimize the expression by refactoring all common prefixes and suffixes.

- The expression is treated as a list of constant-length expressions, from each possible starting expression to the logical end.
- All of these expressions will differ only around places where alternations were present.
- Therefore, everything leading up to the alternation, and everything after the alternation, will be identical for various expressions.
- We can therefore optimize the expression by refactoring all common prefixes and suffixes.

- The expression is treated as a list of constant-length expressions, from each possible starting expression to the logical end.
- All of these expressions will differ only around places where alternations were present.
- Therefore, everything leading up to the alternation, and everything after the alternation, will be identical for various expressions.
- We can therefore optimize the expression by refactoring all common prefixes and suffixes.

- In general, the algorithm will run in $O(n^2)$ time, since every addition of an alternation results in twice as many START nodes.
- This can of course become rapidly intractable, since even having 32 alternations would result in over four billion START nodes.
- While we do perform some optimizations to improve the theoretical running time, it is useful to point out that in the vast majority of practical runs, expressions generally have fewer than twenty alternations.

- In general, the algorithm will run in $O(n^2)$ time, since every addition of an alternation results in twice as many START nodes.
- This can of course become rapidly intractable, since even having 32 alternations would result in over four billion START nodes.
- While we do perform some optimizations to improve the theoretical running time, it is useful to point out that in the vast majority of practical runs, expressions generally have fewer than twenty alternations.

- In general, the algorithm will run in $O(n^2)$ time, since every addition of an alternation results in twice as many START nodes.
- This can of course become rapidly intractable, since even having 32 alternations would result in over four billion START nodes.
- While we do perform some optimizations to improve the theoretical running time, it is useful to point out that in the vast majority of practical runs, expressions generally have fewer than twenty alternations.

- In general, the algorithm will run in $O(n^2)$ time, since every addition of an alternation results in twice as many START nodes.
- This can of course become rapidly intractable, since even having 32 alternations would result in over four billion START nodes.
- While we do perform some optimizations to improve the theoretical running time, it is useful to point out that in the vast majority of practical runs, expressions generally have fewer than twenty alternations.

- The primary space optimization is the shared tail of the core data structure. Outside of pathological cases, this saves considerable storage.
- The alternation processing portion of the algorithm in reality only splits the expression when the two branches of the expression are of unequal length. This reduces the number of paths in many situations.
- Most of the calculations are memoized; that is, they are run only once for any given input for any given position and offset.
- With all of these optimizations taken into account, for most regular expressions in our test data set, runtimes of under three minutes are found (though memory usage is roughly quadratic on the number of operations in the expression).

- The primary space optimization is the shared tail of the core data structure. Outside of pathological cases, this saves considerable storage.
- The alternation processing portion of the algorithm in reality only splits the expression when the two branches of the expression are of unequal length. This reduces the number of paths in many situations.
- Most of the calculations are memoized; that is, they are run only once for any given input for any given position and offset.
- With all of these optimizations taken into account, for most regular expressions in our test data set, runtimes of under three minutes are found (though memory usage is roughly quadratic on the number of operations in the expression).

- The primary space optimization is the shared tail of the core data structure. Outside of pathological cases, this saves considerable storage.
- The alternation processing portion of the algorithm in reality only splits the expression when the two branches of the expression are of unequal length. This reduces the number of paths in many situations.
- Most of the calculations are memoized; that is, they are run only once for any given input for any given position and offset.
- With all of these optimizations taken into account, for most regular expressions in our test data set, runtimes of under three minutes are found (though memory usage is roughly quadratic on the number of operations in the expression).

- The primary space optimization is the shared tail of the core data structure. Outside of pathological cases, this saves considerable storage.
- The alternation processing portion of the algorithm in reality only splits the expression when the two branches of the expression are of unequal length. This reduces the number of paths in many situations.
- Most of the calculations are memoized; that is, they are run only once for any given input for any given position and offset.
- With all of these optimizations taken into account, for most regular expressions in our test data set, runtimes of under three minutes are found (though memory usage is roughly quadratic on the number of operations in the expression).

PERFORMANCE ANALYSIS

Optimizing the Algorithm

- The primary space optimization is the shared tail of the core data structure. Outside of pathological cases, this saves considerable storage.
- The alternation processing portion of the algorithm in reality only splits the expression when the two branches of the expression are of unequal length. This reduces the number of paths in many situations.
- Most of the calculations are memoized; that is, they are run only once for any given input for any given position and offset.
- With all of these optimizations taken into account, for most regular expressions in our test data set, runtimes of under three minutes are found (though memory usage is roughly quadratic on the number of operations in the expression).

- The initial version of the tool was written in Lua.
- However, the tool was quickly rewritten in Erlang.
- Erlang provided a much cleaner method of implementing shared-tail lists, memoization, and bit manipulation.
  - Total implementation is around 1500 lines of heavily-commented Erlang.
  - A large portion of this consists of a hand-crafted PEG library replacing the original YECC based parser.

- The initial version of the tool was written in Lua.
- However, the tool was quickly rewritten in Erlang.
- Erlang provided a much cleaner method of implementing shared-tail lists, memoization, and bit manipulation.
  - Total implementation is around 1500 lines of heavily-commented Erlang.
  - A large portion of this consists of a hand-crafted PEG library replacing the original YECC based parser.

THE IMPLEMENTATION

Basic Structure

- The initial version of the tool was written in Lua.
- However, the tool was quickly rewritten in Erlang.
- Erlang provided a much cleaner method of implementing shared-tail lists, memoization, and bit manipulation.
  - Total implementation is around 1500 lines of heavily-commented Erlang.
  - A large portion of this consists of a hand-crafted PEG library replacing the original YECC based parser.

THE IMPLEMENTATION
Basic Structure

- The initial version of the tool was written in Lua.
- However, the tool was quickly rewritten in Erlang.
- Erlang provided a much cleaner method of implementing shared-tail lists, memoization, and bit manipulation.
  - Total implementation is around 1500 lines of heavily-commented Erlang.
  - A large portion of this consists of a hand-crafted PEG library replacing the original YECC based parser.

THE IMPLEMENTATION

The Future

- Erlang makes it very easy to parallelize computation across multiple threads, processors, and machines.
- The algorithm itself is very amenable to parallelization, and an experimental version of the tool has shown significant speedups.
- This experimental version was also compiled using Erlang's "HiPE" native-code compiler where possible. This resulted in an additional 20% on average speed increase.

- Erlang makes it very easy to parallelize computation across multiple threads, processors, and machines.
- The algorithm itself is very amenable to parallelization, and an experimental version of the tool has shown significant speedups.
- This experimental version was also compiled using Erlang's "HiPE" native-code compiler where possible. This resulted in an additional 20% on average speed increase.

- Erlang makes it very easy to parallelize computation across multiple threads, processors, and machines.
- The algorithm itself is very amenable to parallelization, and an experimental version of the tool has shown significant speedups.
- This experimental version was also compiled using Erlang's "HiPE" native-code compiler where possible. This resulted in an additional 20% on average speed increase.

- While a traditional store-and-forward email server is of course optimal for filtering email, b64re-based signatures on an IPS have been useful in prefiltering email.
- One common use case is filtering obviously malicious email, based on shellcode detection or other parameters, lowering the load on the backend mail server.
- Another interesting application is protecting store-and-forward scanners from themselves. Some systems have bugs that can be triggered by specially formatted emails; an inline IPS can filter these attacks.

- While a traditional store-and-forward email server is of course optimal for filtering email, b64re-based signatures on an IPS have been useful in prefiltering email.
- One common use case is filtering obviously malicious email, based on shellcode detection or other parameters, lowering the load on the backend mail server.
- Another interesting application is protecting store-and-forward scanners from themselves. Some systems have bugs that can be triggered by specially formatted emails; an inline IPS can filter these attacks.

- While a traditional store-and-forward email server is of course optimal for filtering email, b64re-based signatures on an IPS have been useful in prefiltering email.
- One common use case is filtering obviously malicious email, based on shellcode detection or other parameters, lowering the load on the backend mail server.
- Another interesting application is protecting store-and-forward scanners from themselves. Some systems have bugs that can be triggered by specially formatted emails; an inline IPS can filter these attacks.

- While a traditional store-and-forward email server is of course optimal for filtering email, b64re-based signatures on an IPS have been useful in prefiltering email.
- One common use case is filtering obviously malicious email, based on shellcode detection or other parameters, lowering the load on the backend mail server.
- Another interesting application is protecting store-and-forward scanners from themselves. Some systems have bugs that can be triggered by specially formatted emails; an inline IPS can filter these attacks.

DVLabs

COMMON USE CASES

Data Extrusion

- Email is a common source of sensitive data leakage.
- An inline IPS running signatures using this tool can be used to scan for outbound sensitive information, even if an external mail server is used.

- Email is a common source of sensitive data leakage.
- An inline IPS running signatures using this tool can be used to scan for outbound sensitive information, even if an external mail server is used.

- Email is a common source of sensitive data leakage.
- An inline IPS running signatures using this tool can be used to scan for outbound sensitive information, even if an external mail server is used.

FALSE POSITIVES

Detecting the Wrong Thing

- Expressions processed in this way are inherently looser than their unencoded counterparts. Why?
- Recall that the expression is encoded three times, to account for varying offsets from the beginning of input.
- The expression designed to match when its offset is $n + 1$ from the beginning of the input could match input that is actually at a different modular offset.
- This is mitigated somewhat if the expression can be anchored by long constant strings, or if the beginning of input is known.

FALSE POSITIVES

Detecting the Wrong Thing

- Expressions processed in this way are inherently looser than their unencoded counterparts. Why?
- Recall that the expression is encoded three times, to account for varying offsets from the beginning of input.
- The expression designed to match when its offset is $n + 1$ from the beginning of the input could match input that is actually at a different modular offset.
- This is mitigated somewhat if the expression can be anchored by long constant strings, or if the beginning of input is known.

FALSE POSITIVES

Detecting the Wrong Thing

- Expressions processed in this way are inherently looser than their unencoded counterparts. Why?
- Recall that the expression is encoded three times, to account for varying offsets from the beginning of input.
- The expression designed to match when its offset is $n + 1$ from the beginning of the input could match input that is actually at a different modular offset.
- This is mitigated somewhat if the expression can be anchored by long constant strings, or if the beginning of input is known.

FALSE POSITIVES

Detecting the Wrong Thing

- Expressions processed in this way are inherently looser than their unencoded counterparts. Why?
- Recall that the expression is encoded three times, to account for varying offsets from the beginning of input.
- The expression designed to match when its offset is $n + 1$ from the beginning of the input could match input that is actually at a different modular offset.
- This is mitigated somewhat if the expression can be anchored by long constant strings, or if the beginning of input is known.

FALSE POSITIVES

Detecting the Wrong Thing

- Expressions processed in this way are inherently looser than their unencoded counterparts. Why?
- Recall that the expression is encoded three times, to account for varying offsets from the beginning of input.
- The expression designed to match when its offset is $n + 1$ from the beginning of the input could match input that is actually at a different modular offset.
- This is mitigated somewhat if the expression can be anchored by long constant strings, or if the beginning of input is known.

NEWLINE INSERTION

Breaking Up Input

- Most Base64 processors allow the insertion of newlines at arbitrary points of the input.
- The encoded expression can be modified to include an optional newline after every character, though this is expensive.
- Some regular expression engines offer the ability to strip newlines from input, removing the burden on the expression.

- Most Base64 processors allow the insertion of newlines at arbitrary points of the input.
- The encoded expression can be modified to include an optional newline after every character, though this is expensive.
- Some regular expression engines offer the ability to strip newlines from input, removing the burden on the expression.

NEWLINE INSERTION

Breaking Up Input

- Most Base64 processors allow the insertion of newlines at arbitrary points of the input.
- The encoded expression can be modified to include an optional newline after every character, though this is expensive.
- Some regular expression engines offer the ability to strip newlines from input, removing the burden on the expression.

NEWLINE INSERTION

Breaking Up Input

- Most Base64 processors allow the insertion of newlines at arbitrary points of the input.
- The encoded expression can be modified to include an optional newline after every character, though this is expensive.
- Some regular expression engines offer the ability to strip newlines from input, removing the burden on the expression.

SUMMARY

- Using this tool, users can reap several benefits:
  - Pre-filter encoded data streams, lessening the need for expensive decoding.
  - Protect infrastructure that provides decoding services but is itself vulnerable
  - Inspect encoded streams within incurring the overhead of decoding.
- This talk illustrates some interesting challenges when inspecting encoded streams of data, especially in a performance-critical way, and gives some possible solutions.

SUMMARY

- Using this tool, users can reap several benefits:
  - Pre-filter encoded data streams, lessening the need for expensive decoding.
  - Protect infrastructure that provides decoding services but is itself vulnerable
  - Inspect encoded streams within incurring the overhead of decoding.
- This talk illustrates some interesting challenges when inspecting encoded streams of data, especially in a performance-critical way, and gives some possible solutions.

SUMMARY

- Using this tool, users can reap several benefits:
  - Pre-filter encoded data streams, lessening the need for expensive decoding.
  - Protect infrastructure that provides decoding services but is itself vulnerable
  - Inspect encoded streams within incurring the overhead of decoding.
- This talk illustrates some interesting challenges when inspecting encoded streams of data, especially in a performance-critical way, and gives some possible solutions.

FIN

Contact Information

- Rob at Work: rking@tippingpoint.com
- Rob at Home: jking@deadpixi.com