OPTIMIZING CHARACTER-LEVEL TAINT PROPAGATION FOR THE COMMON CASE

Jinghao Yan
Adam Jiang
UC Berkeley

PROJECT BACKGROUND

CHARACTER LEVEL TAINT-PROPAGATION

- Extension of Erika Chin's work on character-level taint tracking
- Detects command injection attacks:
  - SQL Injection
  - XSS
  - Path traversal
  - Shell command injection
- We can limit tracking to Strings

TAINT TRACKING

1. **Source Tainting:** augment the Java Servlets implementation to mark user input as tainted (Tomcat 6)
    1. Form input (GET/POST)
    2. Headers: Cookies, session IDs
2. **Taint Propagation:** augment string classes in the Java library to track taint status (IBM JDK 6)
3. **Sink Checking:** at each sink, detect attacks by checking that control data is not tainted

GOALS

- Focus on taint propagation
  - Source tainting will need minor modifications
  - Details on sink checking are well-documented
- Optimize taint propagation performance for the common case
- But keep the character-level granularity

TAINT PROPAGATION

AUGMENTED CLASSES

- Java.lang.String
- Java.lang.StringBuffer
- Java.lang.StringBuilder
- Javax.security.TaintSet*
  - Java.lang is a protected package

* Our data structure

TAINTSET—OUR DATA STRUCTURE

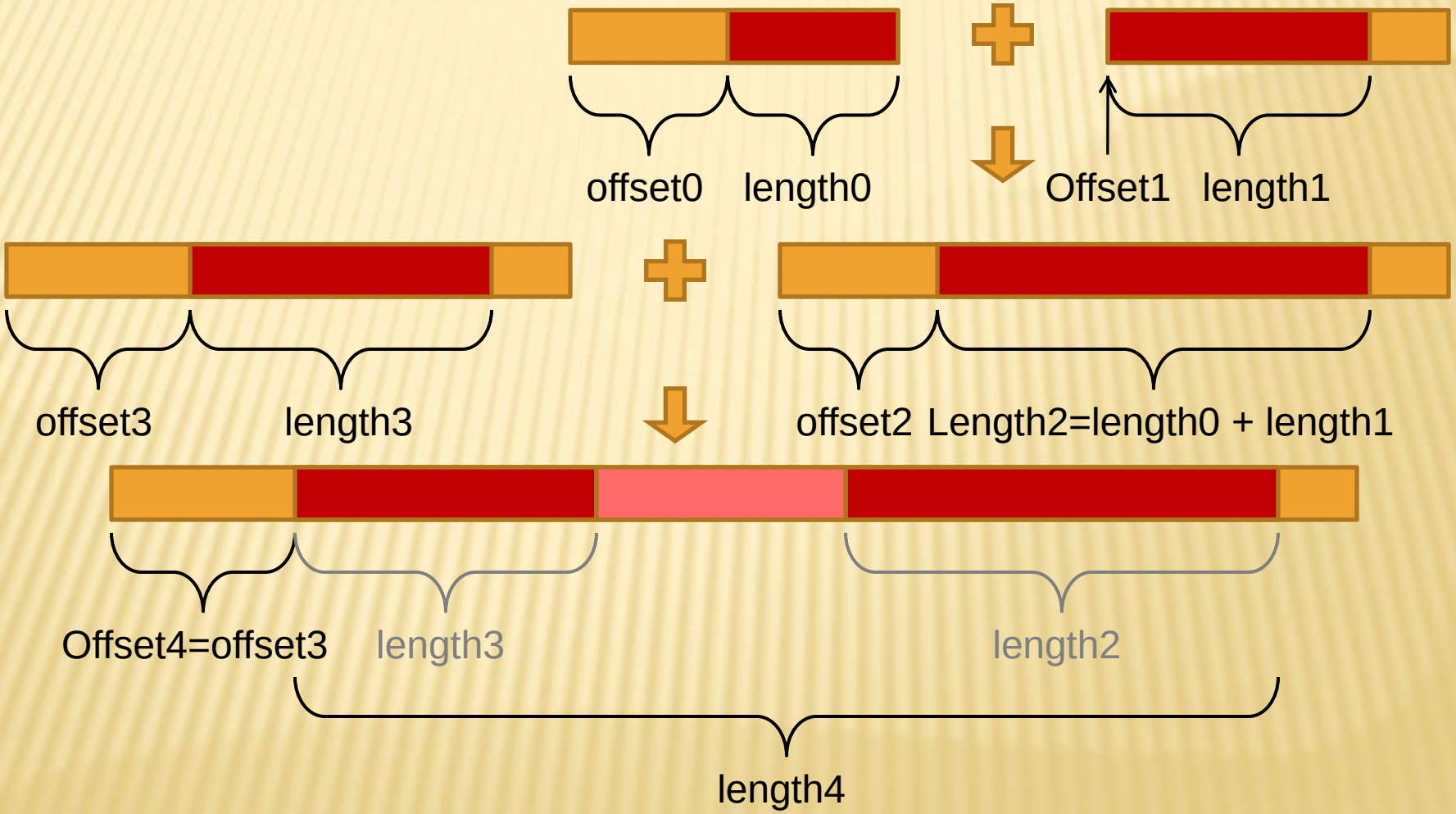
State variables

- Int offset:
  - Index of the first possible tainted character
  - If negative, represents position in taintbits (in bits)
- Int length:
  - Length of the single interval or taintbits (in bits)
- Int[] taintbits:
  - Null if single interval
  - Bits represent taint status otherwise

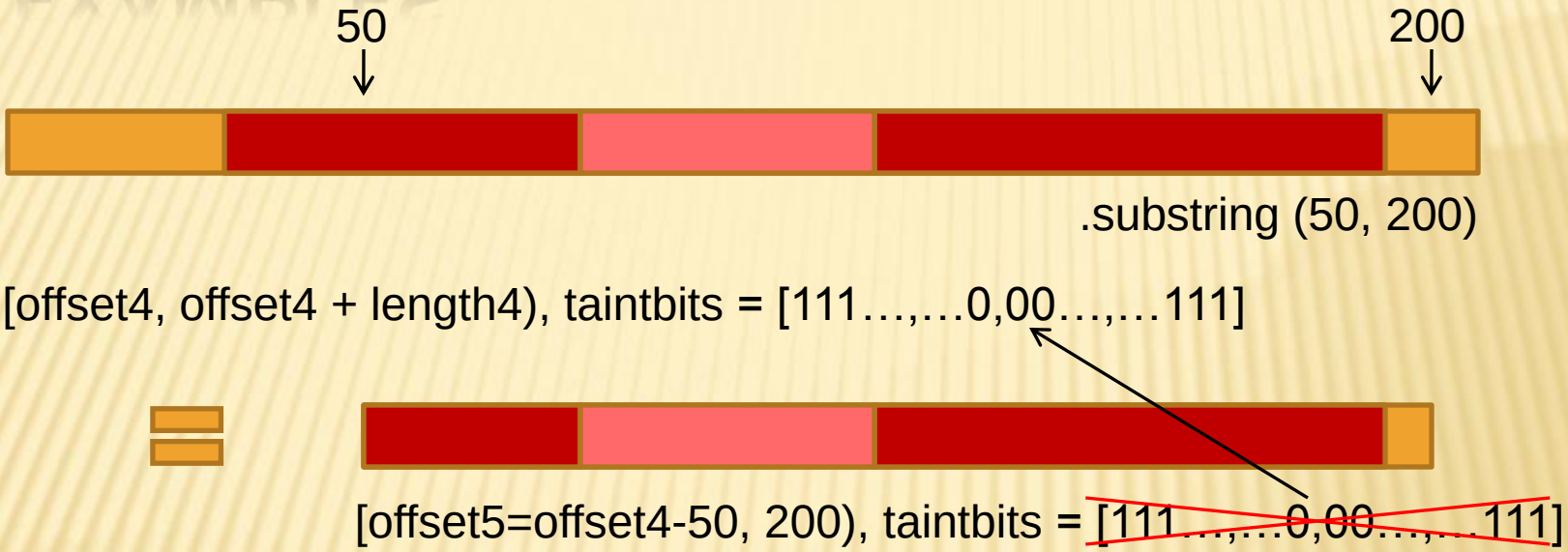
Two representations

1. Single interval:
   - General untrusted input
   - Substrings/combinations of (1)
2. Bit set:
   - Handles higher granularity
   - Copy is created only when necessary—otherwise reused (substring, concatenating strings where at most one is tainted, etc)

EXAMPLES

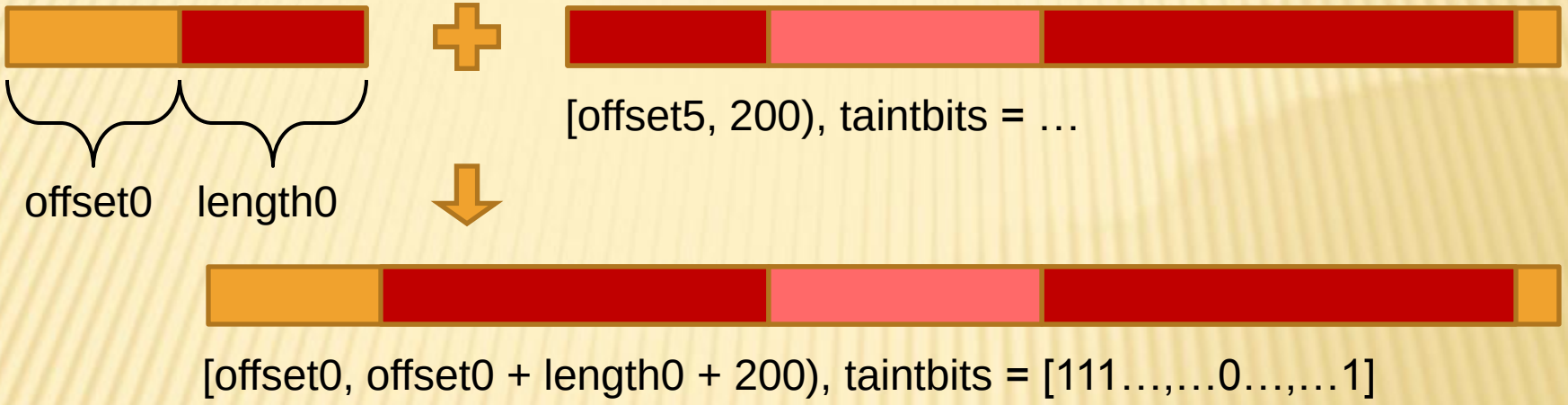

offset0
length0
Offset1
length1
offset3
length3
offset2
Length2=length0 + length1
Offset4=offset3
length3
length2
length4

EXAMPLES



- Strings are immutable, so we can share taintbits
  - StringBuffer/Builder are mutable so substringing requires a copy of relevant portions of taintbits
  - Shared as much as possible (e.g. concatenating with an untainted string)
- Currently generator tests for taint in the region (if not, returns null)

EXAMPLES



Concatenation

1. Creates an integer array large enough to hold 200 + length0 bits
2. Fills the first length0 bits with 1s
3. Copies the [-offset5, -offset5 + 200) bits of the source string's taintbits to the [length0, length0 + 200) bits of the destination string's taintbits

PREVIOUS IMPLEMENTATION

City

B	e	r	k	e	l	e	y
T	T	T	T	T	T	T	T

Punctuation

,	
null	

State

C	A
null	

Temp = Punctuation.concat(State)

,		C	A
null			

Temp2 = City.concat(Temp)

B	e	r	k	e	l	e	y	,		C	A
T	T	T	T	T	T	T	T	F	F	F	F

City.concat(City)

B	e	r	k	e	l	e	y	B	e	r	k	e	l	e	y
T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

OUR IMPLEMENTATION

City

B	e	r	k	e	l	e	y
[0, 8), taintbits = null							

Punctuation

,	
null	

State

C	A
null	

Temp = Punctuation.concat(State)

,		C	A
null			

City.concat(Temp)

B	e	r	k	e	l	e	y	,		C	A

City.concat(City)

B	e	r	k	e	l	e	y	B	e	r	k	e	l	e	y
[0, 16), taintbits = null															

ALTERNATIVE IDEAS

- Pure interval set:
  - Succumbs to pathological cases like

```
for (String s : user_input)
  result += s + ",";
```

- Pure bitmap
  - Handles above case, but generally inefficient
- Single interval + boolean array hybrid
  - Possible consideration
- Use "all tainted" (single interval: [0, ∞)) constant
  - Saves memory (reuses same object), potential operations (e.g. its substring is all tainted)
  - Complicates logic
  - Must change source tainting code

BENCHMARKING

BENCHMARKING

- Many iterations of attempted testing
- Learned about many java pitfalls

BENCHMARKING—ATTEMPT 1

- Used custom class (Stopwatch) to track time using system.nanoTime()
- Recorded start end stop of each test, recorded difference in an array list

```
for i in 0 to TESTLIMIT:
    do initialization not part of the test
    stopwatch.start()
    do test
    stopwatch.stop()
```

BENCHMARKING—ISSUES

- if actual test did not print or otherwise make use of test variable, compiler may optimize out
- system.nano only had 1000ns precision on target machine, meaning unacceptable variance
- ran into recompiliation that showed up as very large time between "locked" code
- did not flush buffers after printing and writing

BENCHMARKING—ATTEMPT 2

- Continue using custom class but try to account for compiler optimizations
- Added warm-up code (many iterations of test code before recording)
- Added extraneous prints outside the "locked" code to ensure compiler would not drop section

BENCHMARKING—ATTEMPT 2

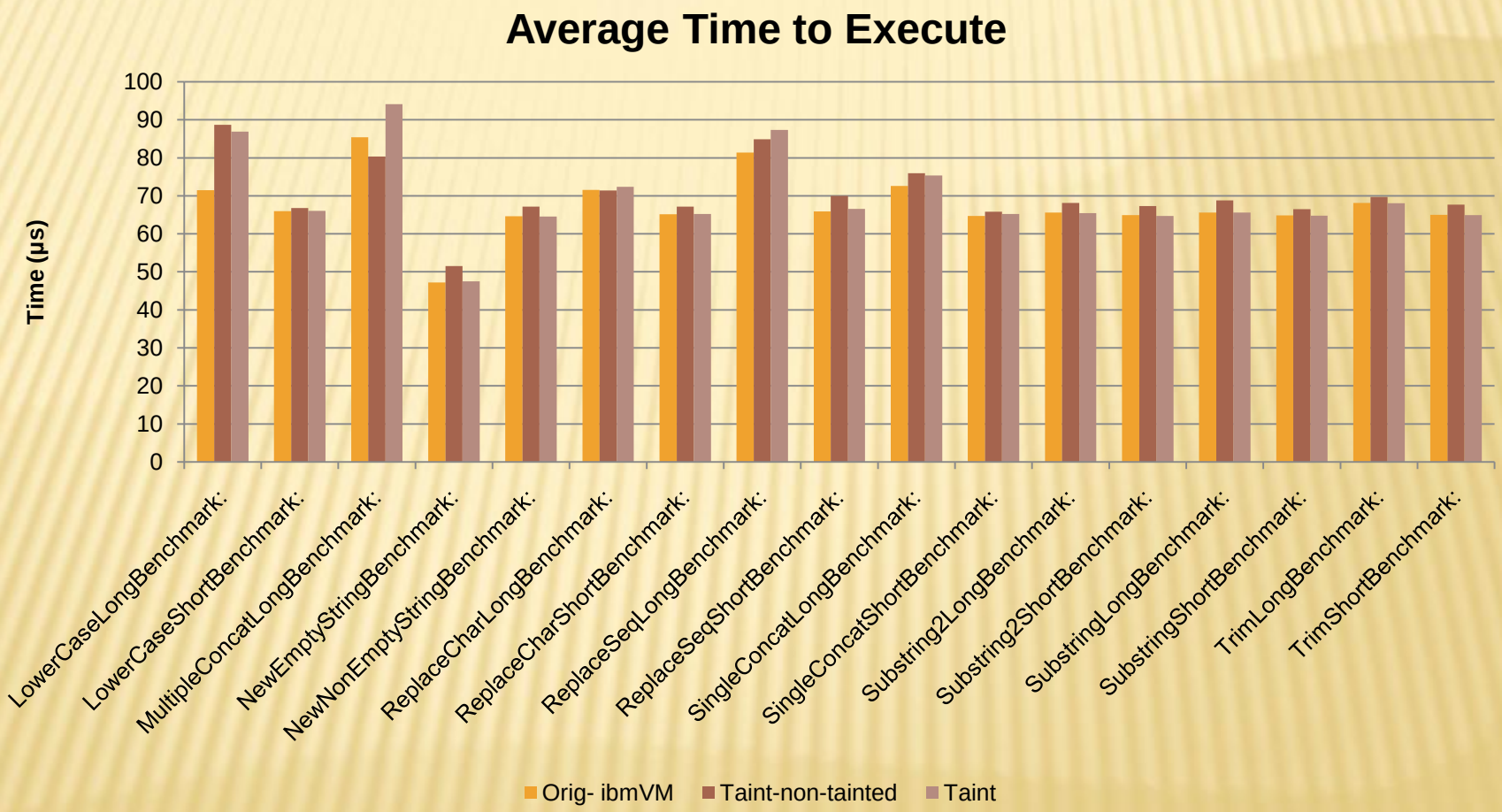
Issues:

- Couldn't accurately determine when code reached steady state (different for different tests)
- Wild outliers and no way to explicitly check for recompilation
- High variance between repeated runs of same code
- Inexplicable test results:
  - toLower on all-upper input faster on 1024 length string than 12

BENCHMARKING—ATTEMPT 3

- Used external framework (Benchmark) that runs code until steady state, restarts with compilation
- Lose a bit of flexibility since all code needs to be wrapped in a Runnable
- All tests include time for a System.out.println(whatever the test output is)
- Tests call runnable 60x in a row, if major time deviation or recompilation occurs, restarts
- Benchmark class outputs to file with a .toString() call, then processed by a script
- Created bash script to run through n iterations of the tests

BENCHMARKING



IMPACT

BENEFITS

- Builds on top of the benefits of Erika Chin's approach: One-time server-side change makes for easy adoption and deployment
- Optimized for the common case:
  - Tainted portions of strings usually contiguous
  - Sink checking of intervals more common
- Memory-efficient
  - Better cache performance
  - Fewer operations to copy and shift integer array

LIMITATIONS

- Some inherited limitations:
  - Serialization does not store taint status
  - May lose taint information via string operations with `char`s/`char` arrays
  - Vulnerable to malicious web developers
  - Format & regular expressions not retrofitted
- Separate data structure with multiple representations introduce complexity

QUESTIONS?