

Pre-Patched Software

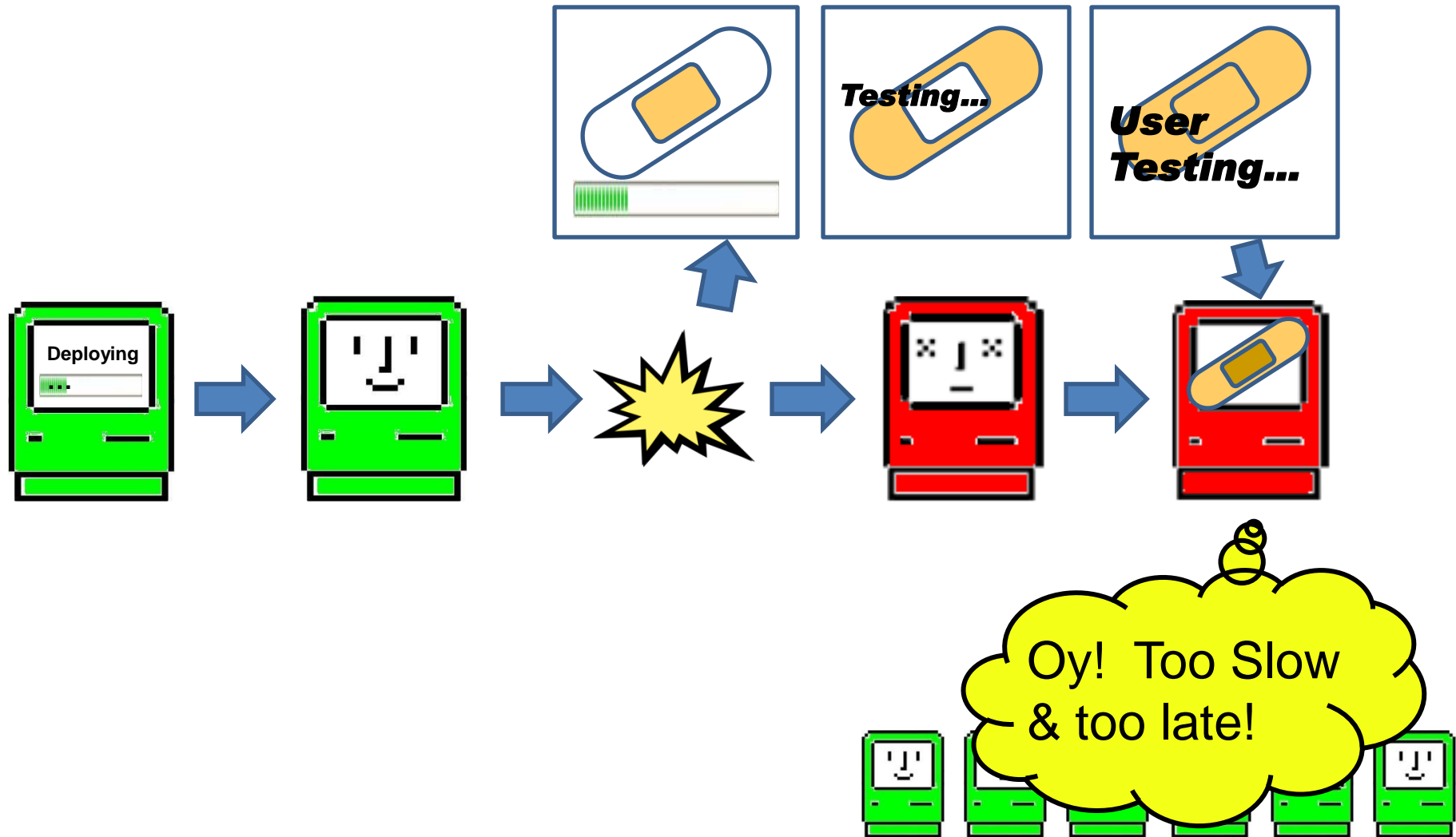
Jianing Guo Jun Yuan Rob Johnson
Stony Brook University

<http://www.splat.cs.sunysb.edu/>

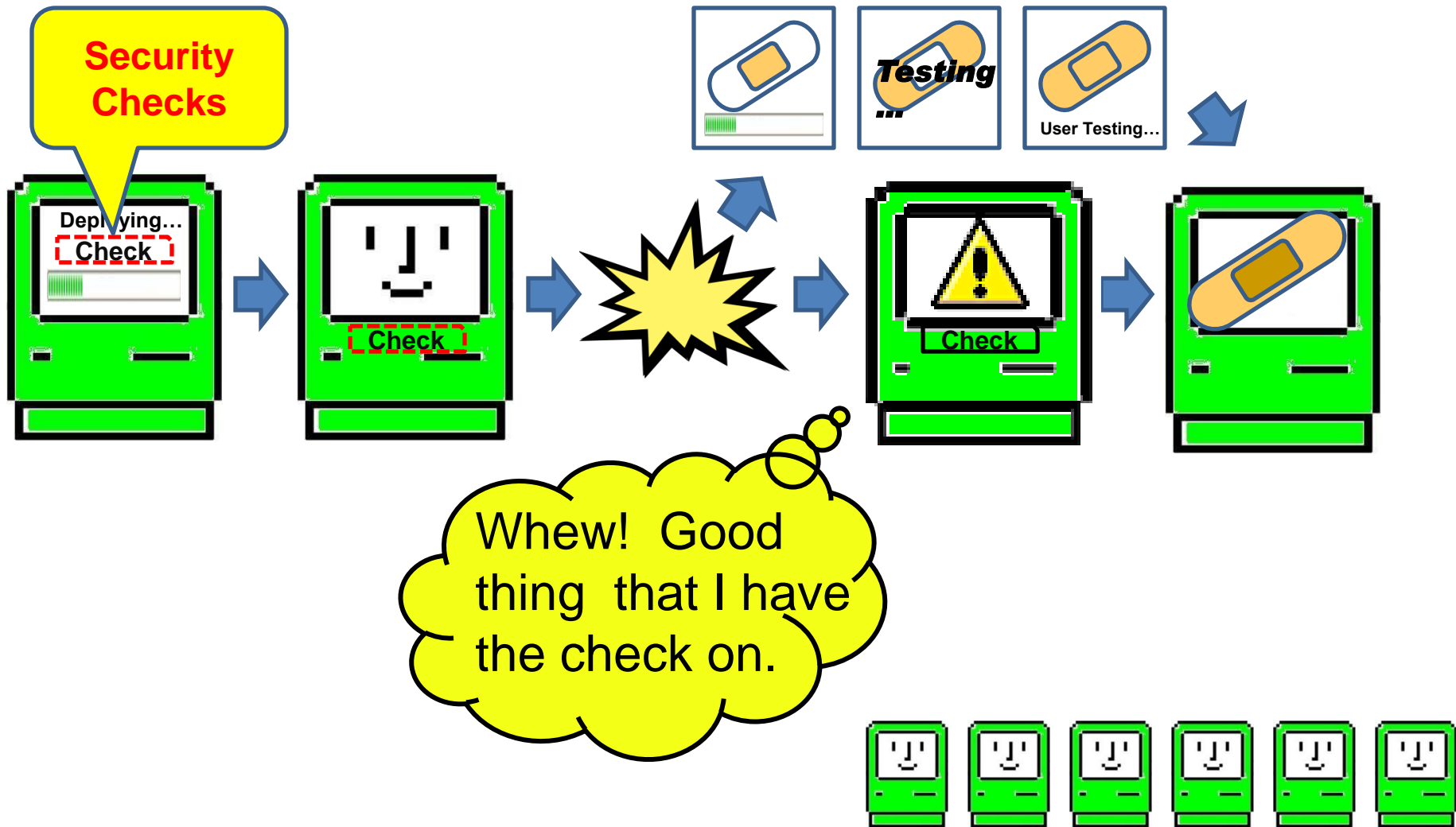
Bugs in Deployed Software

- The problem with patches:
 - ◆ Slow and error-prone to develop
 - ◆ Long “window of vulnerability” that exposes users to a possible “zero day exploit”
- The problem with run-time checks
 - ◆ High overhead
 - ◆ Compatibility issues
- Pre-Patched Software
 - ◆ Uses latent run-time checks
 - ◆ Low run-time overhead
 - ◆ Rapid response to new vulnerabilities
 - ◆ Backwards compatible

Zero Day Exploit Problem



Pre-Patched Software



Benefits

- Provides immediate response to vulnerabilities
- Prevents “zero day exploit”
- Users don’t pay a visible overhead until it becomes necessary
- Shipping instrumented binaries allows users to test in advance

Prototype: Memsafe

- Checks against bounds violations
- Based on Jones & Kelly's [Jones 97] approach to C bounds checking
- Implemented using CIL [Necula 02] platform

Memsafe Example

```
void foo () {  
    int arr[5];  
    B_arr = Register(arr);  
    int n = 1;  
    B_arr = LookUp(arr + 0);  
    Check(arr+0, B_arr );  
    arr[0] = n;  
    n = bar(n, arr, B_arr );  
}  
  
int bar(int n, int*a, bounds B_a ) {  
    int i, s = 0;  
    Check(a+0, B_a );  
    s = a[0];  
    for( i = 0; i <= n; i++ ) {  
        a++;  
    }  
    return s;  
}
```

Not a
Problem

- Register only necessary variables
- Caching bounds info
- Bounds passing across functions.
- Support manipulation for OOB ptrs

Memsafe Optimizations

- Bounds caching
- Bounds passing
- Loop optimization
- Static check elimination

Run-time Check Activation

- Selectively turn on checks – reduces patch overhead
- Instrumentation dependency -- enables metadata maintenance
- Fast path/Slow path – saves time on branch checking

Not memsafe specific

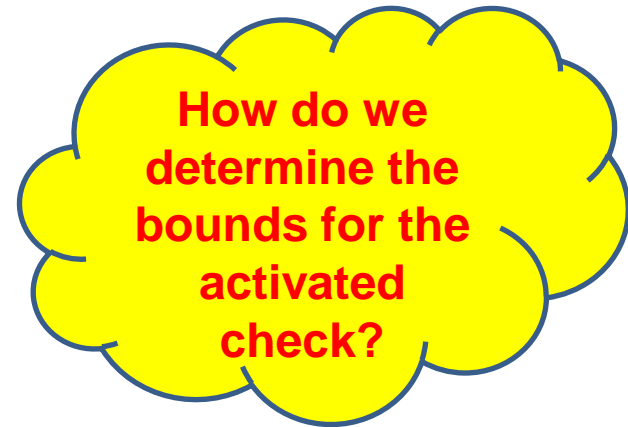
Selective Check Activation

| | | | | | | | |
|---|---|---|---|---|---|---|-----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | ... |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | ... |

```
void foo () {  
  int arr[5];  
  1 B_arr = Register(arr);  
  int n = 1;  
  2 Check(arr+0, B_arr );  
  arr[0] = n;  
  n = bar (arr, B-a );  
}  
int bar (int n, int*a, bounds B-a )  
{  
  int i, s = 0;  
  3 Check(a+0, B-a );  
  s = a[0];  
  for( i = 0; i <= n; i++ ) {  
    a++;  
  }  
  return s;  
}
```

- Checks can be activated independently based on the bit map.

Dependencies



```
void foo () {  
  int arr[5];  
  1 B_arr = Register(arr);  
  int i = 1;  
  2 Check(a+0, B_arr);  
  arr[0] = n;  
  n = bar(arr, B-arr);  
}  
int bar (int n, int*a, bounds B-a )  
{  
  int i, s = 0;  
  3 Check(a+0, B-a);  
  s = a[0];  
  for( i = 0; i <= n; i++ ) {  
    a++;  
  }  
  return s;  
}
```

- Dependency within a single function
- Dependency across functions

Fast-Path/Slow-Path



if (any active checks)

How to reduce the number of checks performed at run time?

Slow Path

```
{  
  int arr[5];  
  B_arr = Register(arr);  
  int n = 1;  
  Check(arr+0, B_arr);  
  arr[0] = n;  
  n = bar (arr, B-a );  
}
```

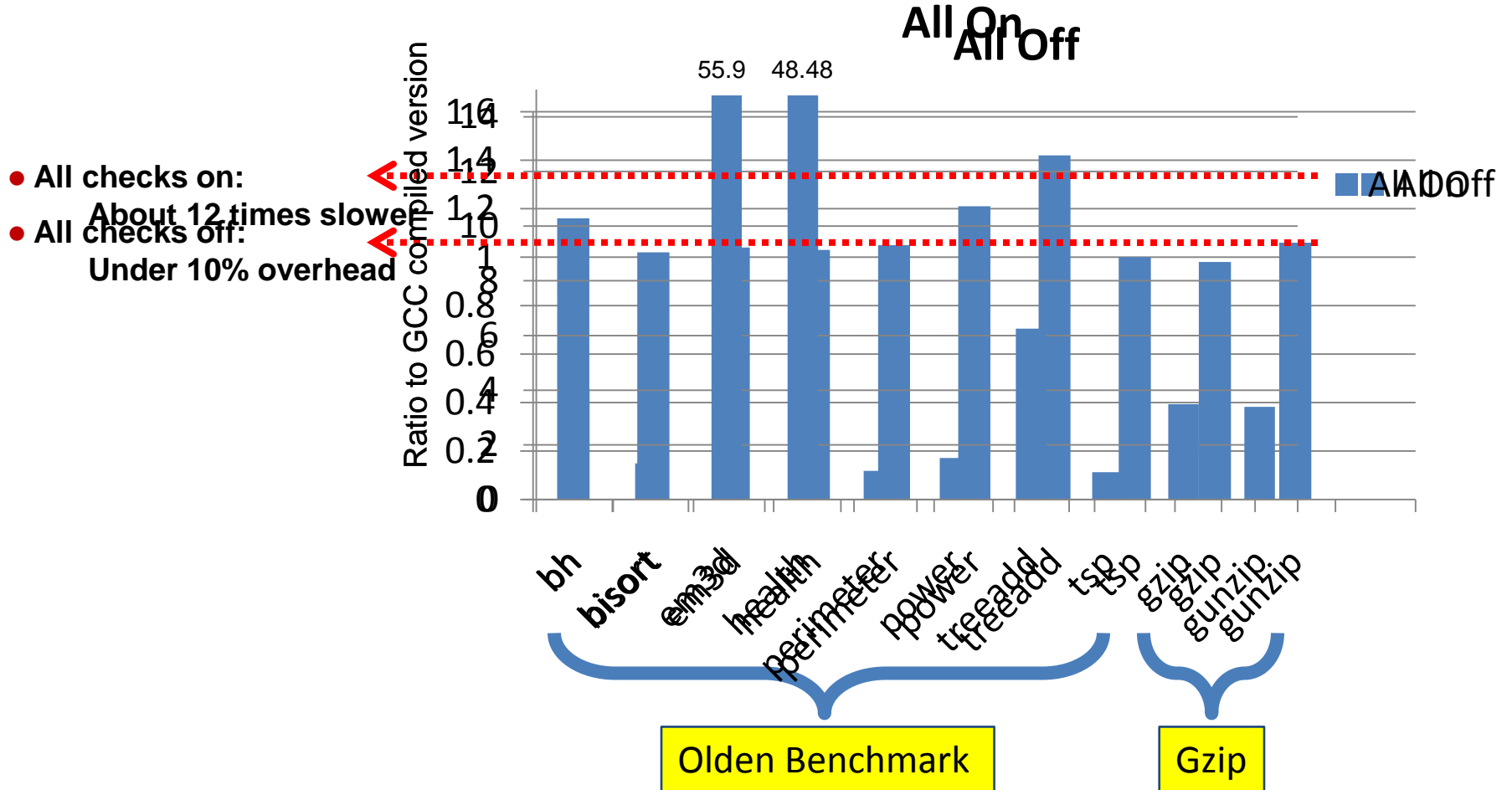
Fast Path

```
{  
  int arr[5];  
  
  int n = 1;  
  
  arr[0] = n;  
  n = bar (arr, B-a );  
}
```

Performance Evaluation

- Three scenarios:
 - ◆ All checks off (common case)
 - ◆ One check on (occasional case)
 - ◆ All checks on (only for testing)
- Benchmark programs:
 - ◆ Gzip and Gunzip
 - ◆ Olden Benchmark [Rogers 95, Carlisle 95]

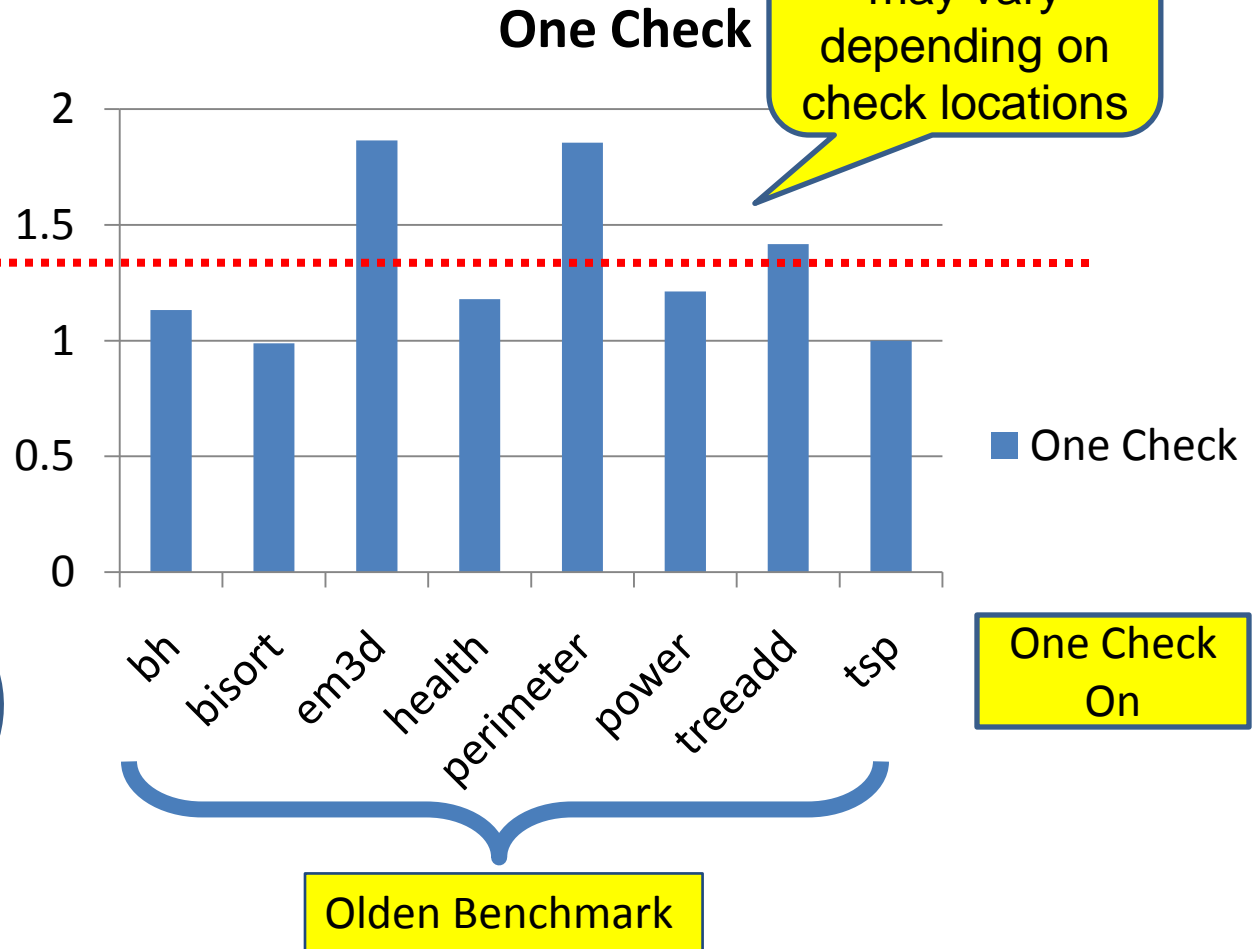
Results



Results

● One Check On:
About 33% overhead

Overhead is negligible in comparison to all checks on



Limitations

- Not as efficient & complete as patches
- Depends on compiler auto-generation
- Only applicable to low level security bugs

Conclusion

- Pre-patched software provides immediate response to vulnerabilities
- Latent run-time checks incur low overhead while providing full coverage
- Pre-patched software makes code transformations usable by reducing overheads to a fraction

Q&A

Pre-Patched Software

Jianing Guo Jun Yuan Rob Johnson
Stony Brook University

<http://www.splat.cs.sunysb.edu/>