Putting the Checks into Checked C

Archibald Samuel Elliott

Quals - 31st Oct 2017

Added Runtime **Bounds Checks** to the **Checked C** Compiler

C Extension for
Spatial Memory Safety

Added Runtime **Bounds Checks** to the **Checked C** Compiler

C Extension for Spatial Memory Safety

Bounds Propagation Algorithm

Added Runtime **Bounds Checks** to the **Checked C** Compiler

C Extension for Spatial Memory Safety

Bounds Propagation Algorithm

Added Runtime **Bounds Checks** to the **Checked C** Compiler

First Evaluation of Checked C

C Extension for Spatial Memory Safety

Bounds Propagation Algorithm

Added Runtime **Bounds Checks** to the **Checked C** Compiler

First Evaluation of Checked C

In Our Clang/LLVM Fork

Motivation

- Buffer overflows account for 10-16% of Vulnerabilities
- Affecting:
  - Operating Systems
  - Web Browsers
  - OpenSSL
  - Programming Language Implementations

Related Work

Approach	Examples
More Better Types	Deputy; CCured
Runtime Representations	CCured; SoftBound
Static Analysis Annotations	SAL
Instrumentation (for Fuzzing)	ASan; UBSan

- Checked C

C Pointers

```
T* val = malloc(sizeof(T));
```

Singleton Pointer

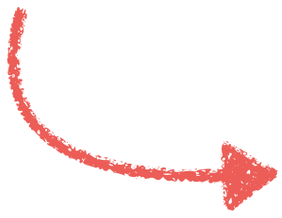
```
T* vals  = calloc(n, sizeof(T));

T vals[N] = { ... };
```

Array Pointer

C Pointers

```
T* val = malloc(sizeof(T));
```

Singleton Pointer



```
ptr<T> val = malloc(sizeof(T));
```

```
T* vals  = calloc(n, sizeof(T));

T vals[N] = { ... };
```

Array Pointer

C Pointers

```
T* val = malloc(sizeof(T));
```

Singleton Pointer

```
ptr<T> val = malloc(sizeof(T));
```

```
T* vals  = calloc(n, sizeof(T));

T vals[N] = { ... };
```

Array Pointer

```
array_ptr<T> vals  = calloc(n, sizeof(T));

T vals checked[N] = { ... };
```

Bounds Declarations

Declaration	Access Invariant
`array_ptr<T> p : bounds(l, u)`	`p' != NULL && l <= p' < u`

Bounds Declarations

Declaration	Access Invariant
`array_ptr<T> p : bounds(l, u)`	`p' != NULL && l <= p' < u`
`array_ptr<T> p : count(n)`	`p' != NULL && p <= p' < p + n`
`array_ptr<T> p : byte_count(n)`	`p' != NULL && p <= p' < (char*)p + n`

Bounds Declarations

Declaration	Access Invariant
`array_ptr<T> p : bounds(l, u)`	`p' != NULL && l <= p' < u`
`array_ptr<T> p : count(n)`	`p' != NULL && p <= p' < p + n`
`array_ptr<T> p : byte_count(n)`	`p' != NULL && p <= p' < (char*)p + n`

Expressions in **`bounds(l, u)`** must be non-modifying

- No Assignments or Increments/Decrements
- No Calls

Interoperation with Unchecked Code

- Annotate Unchecked Pointers with Bounds

```
T* val : itype(ptr<T>) = malloc(sizeof(T));

T* vals : count(n) = calloc(n, sizeof(T));
```

- Checked and Unchecked Scopes

```
checked { ... }
unchecked { ... }

checked   int my_func1(int s, int* : count(s)) {
unchecked int my_func2(int s, int* : count(s)) {
```

Soundness

Well-Foundedness:

- All bounds expressions for variables or data are defined and a sub-range of their objects in memory; and,

- All non-null pointers of type T with bounds must point to an object in memory of type T.

Soundness:

- Assuming Memory & Program are Well-founded on entry of Checked scope

- Evaluation Preserves Well-foundedness; and

- Memory Reads and Writes Preserve Well-Foundedness

- Checked C

- Checked C

- **Example**

```
bool echo(
   int16_t user_length,
   size_t  user_payload_len,
   char   *user_payload,
   resp_t *resp) {
```

Copy data
from `user_payload`
into new buffer in
`resp` object

```
}
```

```
bool echo(
   int16_t user_length,
   size_t  user_payload_len,
   char    *user_payload,
   resp_t *resp) {
```

Copy data from `user_payload` into new buffer in `resp` object

```
}
```

`user_length` is provided by user

`user_payload_len` is from the parser

```
bool echo(
   int16_t user_length,
   size_t  user_payload_len,
   char   *user_payload,
   resp_t *resp) {
```

Copy data from `user_payload` into new buffer in `resp` object

```
}
```

`user_length` is provided by user

`user_payload_len` is from the parser

```
typedef struct {
   size_t payload_len;
   char  *payload;
   // ...
} resp_t;
```

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  char   *user_payload,
  resp_t *resp) {

  char *resp_data = malloc(user_length);

  resp->payload     = resp_data;
  resp->payload_len = user_length;
```

Copy data from `user_payload` into new buffer in `resp` object

```
}
```

`user_length` is provided by user

`user_payload_len` is from the parser

```
typedef struct {
  size_t payload_len;
  char  *payload;
  // ...
} resp_t;
```

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  char   *user_payload,
  resp_t *resp) {

  char *resp_data = malloc(user_length);

  resp->payload     = resp_data;
  resp->payload_len = user_length;

}
```

Copy data from `user_payload` into new buffer in `resp` object

`malloc` could fail

`user_length` is provided by user

`user_payload_len` is from the parser

```
typedef struct {
  size_t payload_len;
  char  *payload;
  // ...
} resp_t;
```

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  char   *user_payload,
  resp_t *resp) {

  char *resp_data = malloc(user_length);

  resp->payload     = resp_data;
  resp->payload_len = user_length;

  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Copy data from `user_payload` into new buffer in `resp` object

`user_length` is provided by user

`user_payload_len` is from the parser

```
typedef struct {
  size_t payload_len;
  char  *payload;
  // ...
} resp_t;
```

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  char   *user_payload,
  resp_t *resp) {

  char *resp_data = malloc(user_length);

  resp->payload     = resp_data;
  resp->payload_len = user_length;

  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Copy data from `user_payload` into new buffer in `resp` object

`user_length` is ... `user_payload_len` is ...

`user_length` could be larger than `user_payload_len`

```
typedef struct {
  size_t payload_len;
  char  *payload;
  // ...
} resp_t;
```

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  char   *user_payload,
  resp_t *resp) {

  char *resp_data = malloc(user_length);

  resp->payload     = resp_data;
  resp->payload_len = user_length;

  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Copy data from `user_payload` into new buffer in `resp` object

`user_length` is provided by user

`user_payload_len` is from the parser

```
typedef struct {
  size_t payload_len;
  char  *payload;
  // ...
} resp_t;
```

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload,
  ptr<resp_t> resp) {

  array_ptr<char> resp_data = malloc(user_length);

  resp->payload     = resp_data;
  resp->payload_len = user_length;

  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Step 1:
Manually
Convert to
Checked Types

```
typedef struct {
  size_t payload_len;
  array_ptr<char> payload;
  // ...
} resp_t;
```

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload : count(user_payload_len),
  ptr<resp_t> resp) {

  array_ptr<char> resp_data : count(user_length) = malloc(user_length)

  resp->payload     = resp_data;
  resp->payload_len = user_length;

  // memcpy(resp->payload, user_payl
  for (size_t i = 0; i < user_length
    resp->payload[i] = user_payload[
  }
  return true;
}
```

Step 2:
Manually
Add Bounds
Declarations

```
typedef struct {
  size_t payload_len;
  array_ptr<char> payload : count(payload_len);
  // ...
} resp_t;
```

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload : cou
  ptr<resp_t> resp) {

  array_ptr<char> resp_data : count(user_length) = malloc(user_length)

  dynamic_check(resp != NULL);
  resp->payload     = resp_data;
  resp->payload_len = user_length;

  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    dynamic_check(user_payload != NULL);
    dynamic_check(user_payload <= &user_payload[i]);
    dynamic_check(&user_payload[i] < user_payload + user_payload_len);
    dynamic_check(resp->payload != NULL);
    dynamic_check(resp->payload <= &resp->payload[i]);
    dynamic_check(&resp->payload[i] < resp->payload + resp->payload_le
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Step 3:
Compiler Inserts
Checks Automatically

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload : cou
  ptr<resp_t> resp) {

  array_ptr<char> resp_data : count(user_length) = malloc(user_length)

  dynamic_check(resp != NULL);
  resp->payload     = resp_data;
  resp->payload_len = user_length;

  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    dynamic_check(user_payload != NULL);
    dynamic_check(user_payload <= &user_payload[i]);
    dynamic_check(&user_payload[i] < user_payload + user_payload_len);
    dynamic_check(resp->payload != NULL);
    dynamic_check(resp->payload <= &resp->payload[i]);
    dynamic_check(&resp->payload[i] < resp->payload + resp->payload_le
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Step 3:
Compiler Inserts
Checks Automatically

No Memory Disclosure

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload : cou
  ptr<resp_t> resp) {

  array_ptr<char> resp_data : count(user_length) = malloc(user_length)

  dynamic_check(resp != NULL);
  resp->payload     = resp_data;
  resp->payload_len = user_length;

  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    dynamic_check(user_payload != NULL);
    dynamic_check(user_payload <= &user_payload[i]);
    dynamic_check(&user_payload[i] < user_payload + user_payload_len);
    dynamic_check(resp->payload != NULL);
    dynamic_check(resp->payload <= &resp->payload[i]);
    dynamic_check(&resp->payload[i] < resp->payload + resp->payload_le
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Step 3:
Compiler Inserts
Checks Automatically

`malloc` now checked

No Memory Disclosure

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload : cou
  ptr<resp_t> resp) {

  array_ptr<char> resp_data : count(user_length) = malloc(user_length)

  dynar
  resp-
  resp-

  // me
  for
    dyn
    dyn
    dyn
    dyn
    dynamic_check(resp->payload <= &resp->payload[i]);
    dynamic_check(&resp->payload[i] < resp->payload + resp->payload_le
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Step 3:
Compiler Inserts
Checks Automatically

Code **Not** Bug-Free:
Will signal run-time error if either

- `malloc(user_length)` fails
- `user_length > user_payload_len`

But: Vulnerable Executions Prevented

No Memory Disclosure

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload : cou
  ptr<resp_t> resp) {

  array_ptr<char> resp_data : count(user_length) = malloc(user_length)

  dynamic_check(resp != NULL);
  resp->payload     = resp_data;
  resp->payload_len = user_length;

  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    dynamic_check(user_payload != NULL);
    dynamic_check(user_payload <= &user_payload[i]);
    dynamic_check(&user_payload[i] < user_payload + user_payload_len);
    dynamic_check(resp->payload != NULL);
    dynamic_check(resp->payload <= &resp->payload[i]);
    dynamic_check(&resp->payload[i] < resp->payload + resp->payload_le
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Step 3:
Compiler Inserts
Checks Automatically

`malloc` now checked

No Memory Disclosure

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload : c
  ptr<resp_t> resp) {

  array_ptr<char> resp_data : count(user_length) = malloc(user_length)

  dynamic_check(resp != NULL);
  resp->payload     = resp_data;
  resp->payload_len = user_length;

  dynamic_check(user_payload != NULL);
  dynamic_check(resp->payload != NULL);
  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    dynamic_check(i <= user_payload_len);
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Step 4:
Restrictions on bounds expressions may allow removal

```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload : c
  ptr<resp_t> resp) {

  array_ptr<char> resp_data : count(user_length) = malloc(user_length)

  dynamic_check(resp != NULL);
  resp->payload     = resp_data;
  resp->payload_len = user_length;

  dynamic_check(user_payload != NULL);
  dynamic_check(resp->payload != NULL);
  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    dynamic_check(i <= user_payload_len);
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Step 4:
Restrictions on bounds expressions may allow removal

No Memory Disclosure

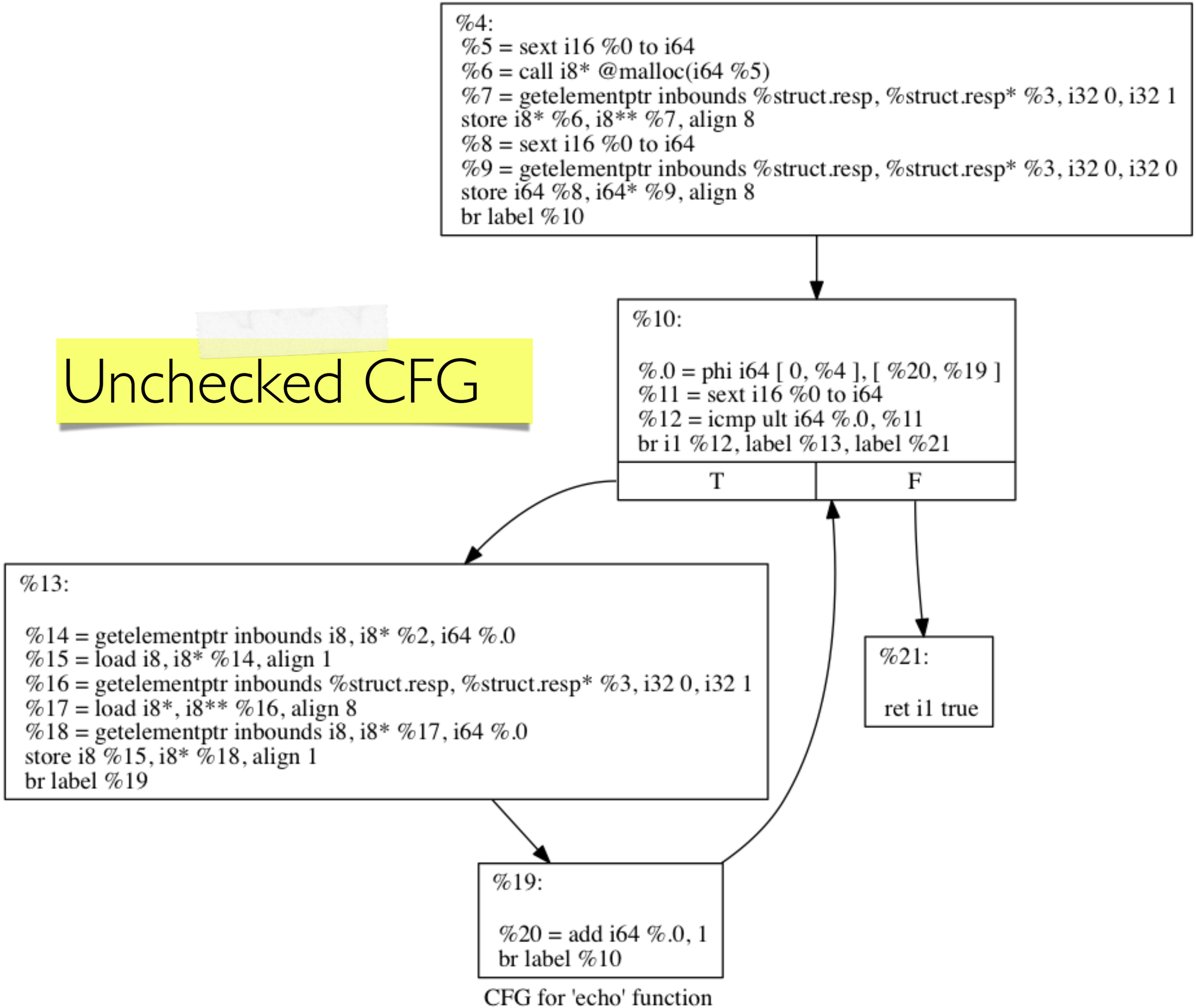
```
bool echo(
  int16_t user_length,
  size_t  user_payload_len,
  array_ptr<char> user_payload : c
  ptr<resp_t> resp) {

  array_ptr<char> resp_data : count(user_length) = malloc(user_length)

  dynamic_check(resp != NULL);
  resp->payload     = resp_data;
  resp->payload_len = user_length;

  dynamic_check(user_payload != NULL);
  dynamic_check(resp->payload != NULL);
  // memcpy(resp->payload, user_payload, user_length)
  for (size_t i = 0; i < user_length; i++) {
    dynamic_check(i <= user_payload_len);
    resp->payload[i] = user_payload[i];
  }
  return true;
}
```

Step 4:
Restrictions on bounds expressions may allow removal

`malloc` still checked

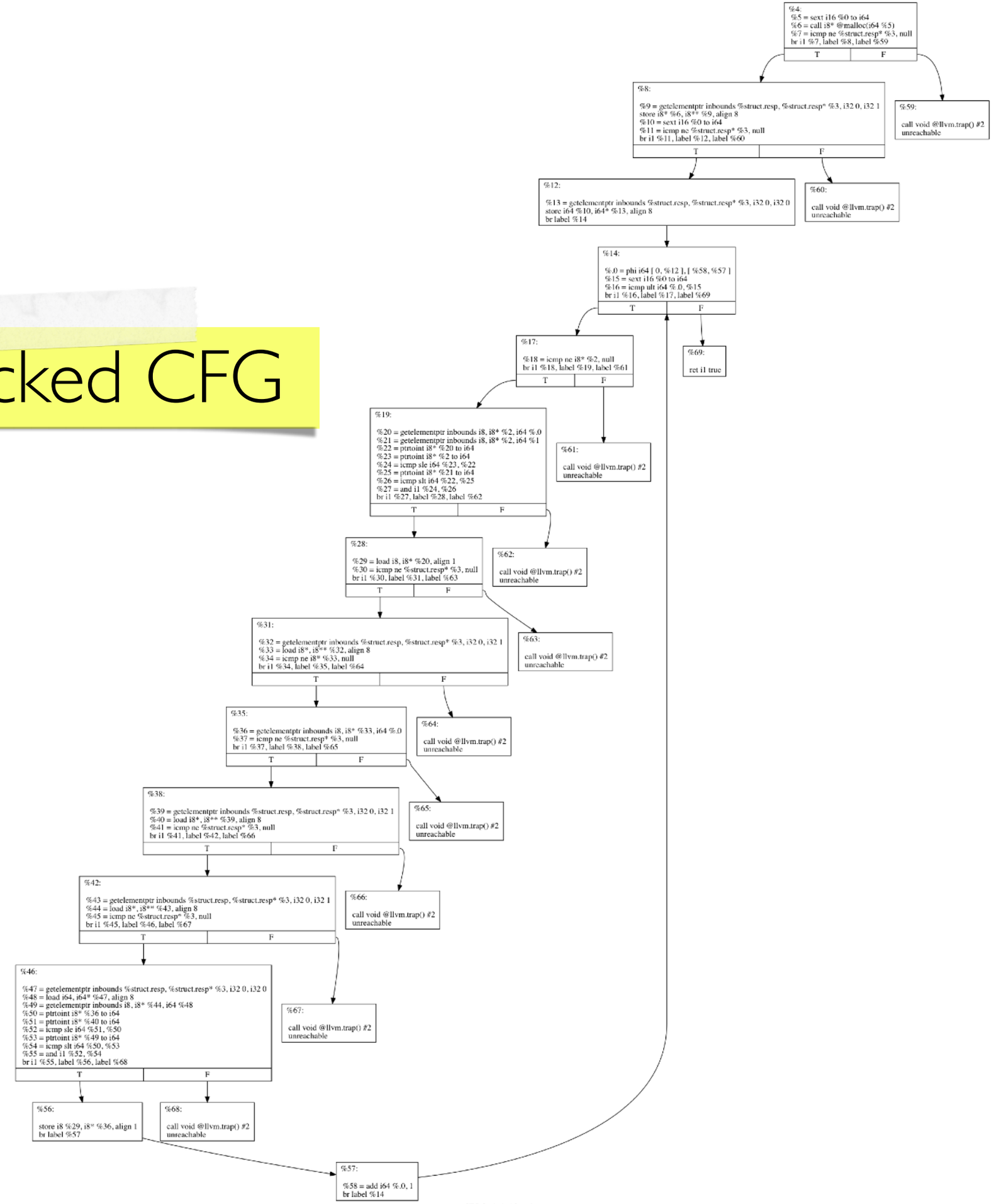
No Memory Disclosure

Unchecked CFG



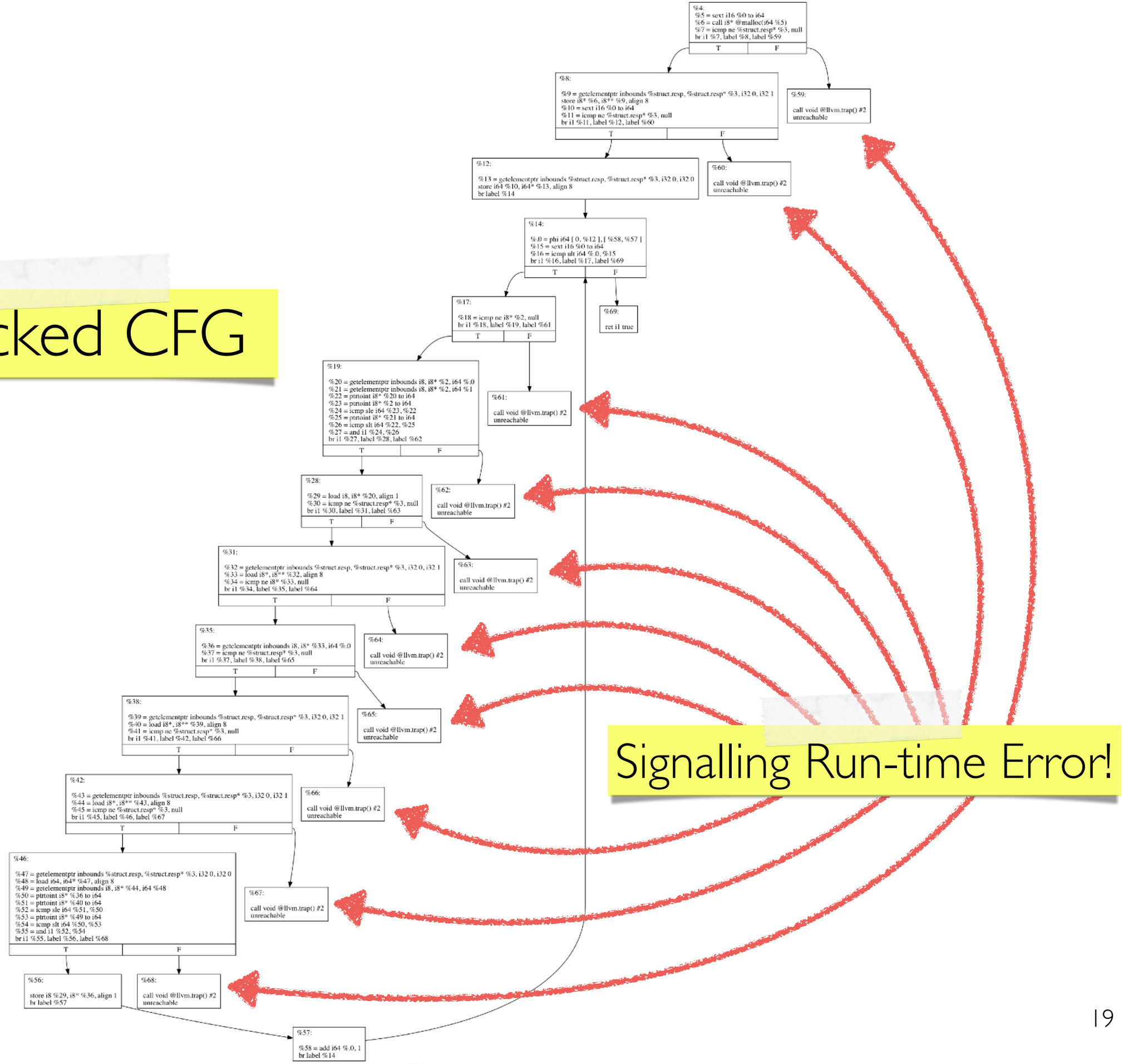
CFG for 'echo' function

Checked CFG



CFG for 'echo' function

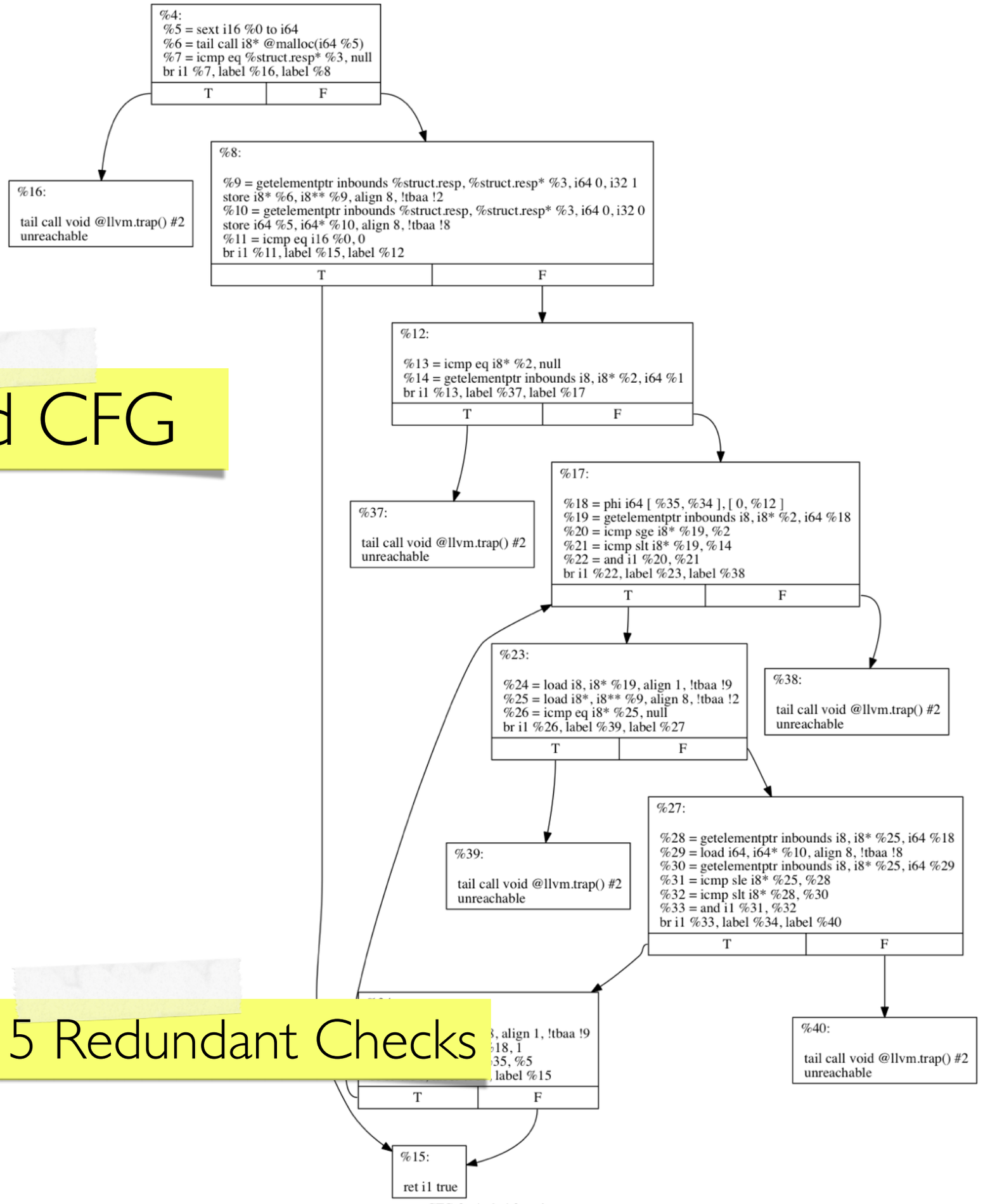
Checked CFG



CFG for 'echo' function

Signalling Run-time Error!

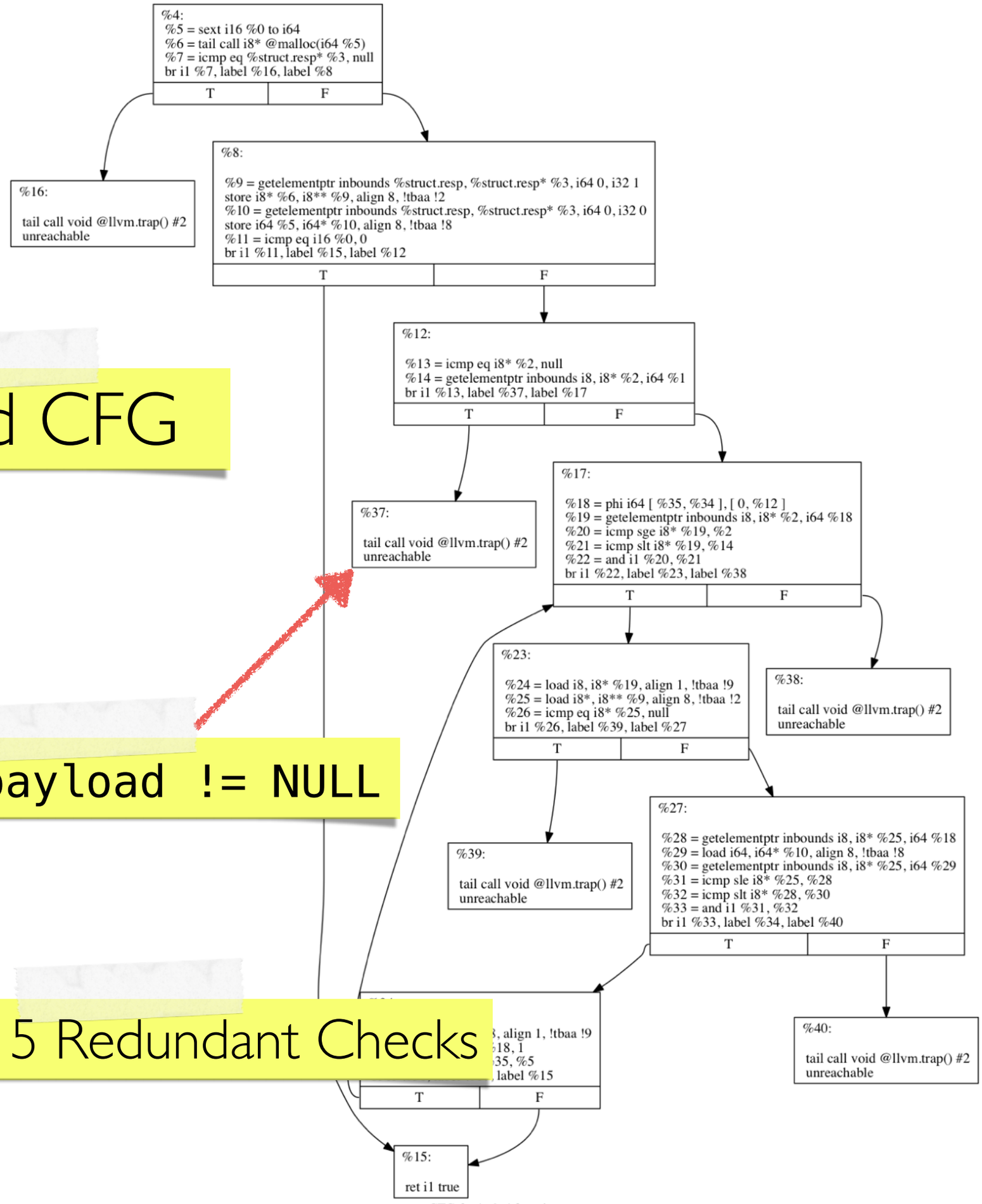
Checked CFG



CFG for 'echo' function

Removed 5 Redundant Checks

Checked CFG



CFG for 'echo' function

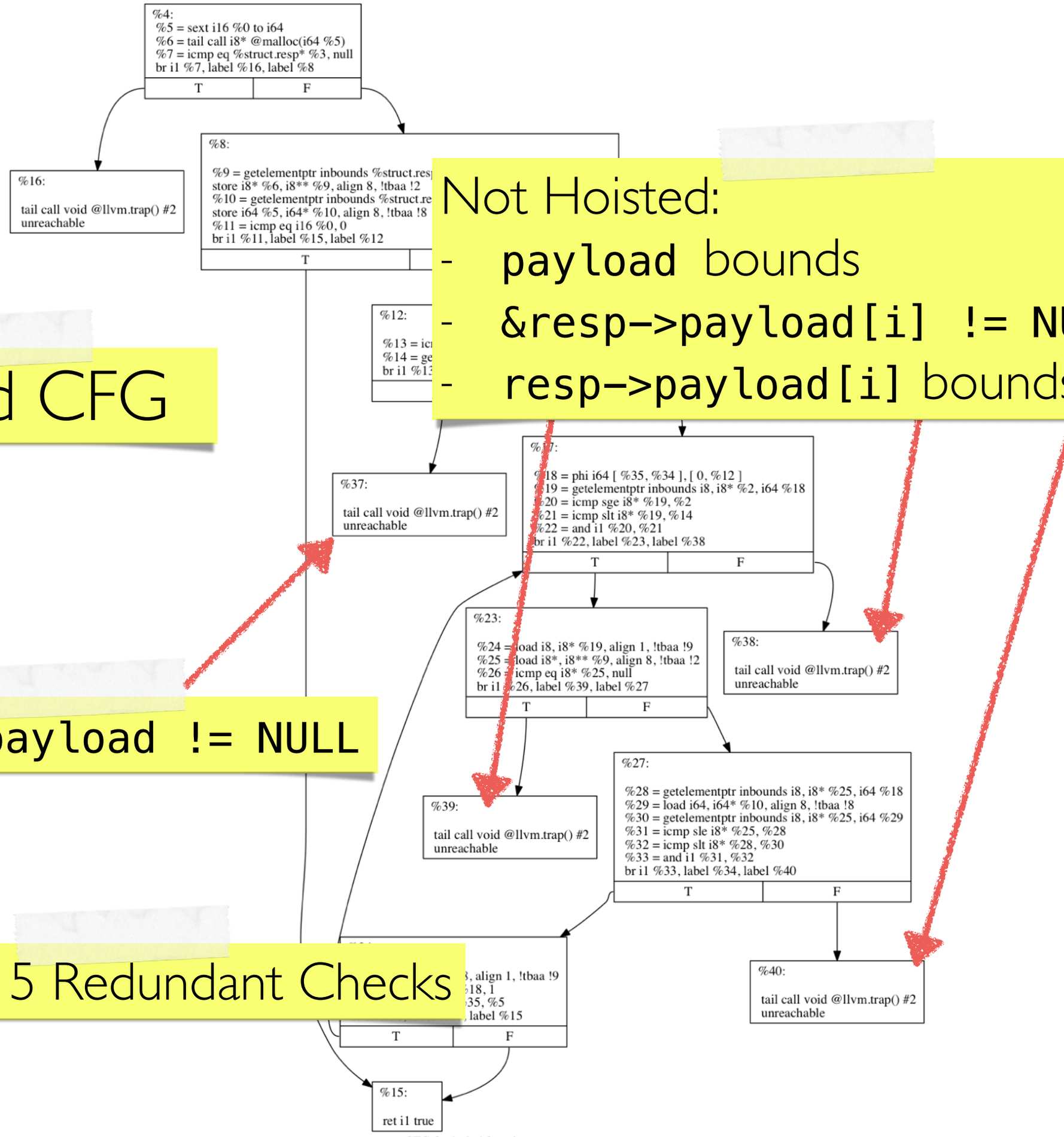
Hoisted: `payload != NULL`

Removed 5 Redundant Checks

Checked CFG

Not Hoisted:

- `payload` bounds
- `&resp->payload[i] != NULL`
- `resp->payload[i]` bounds



CFG for 'echo' function

Hoisted: `payload != NULL`

Removed 5 Redundant Checks

- Checked C

- **Example**

- Checked C
- Example
- Dynamic Checks

C Expression Semantics

- Expressions evaluate to Lvalues or Values
- Lvalues are locations of objects in memory
- Lvalues are used to read or write memory
- Values are integers, floats, pointers

Lvalue Expressions

Lvalue Expressions

Creation
`variable`
`*expr`
`expr1[expr2]`
`lvalue_expr.field`
`expr->field`

Lvalue Expressions

Creation	Use
`variable`	`lvalue_expr = expr`
`*expr`	`&lvalue_expr`
`expr1[expr2]`	`lvalue_expr += expr`
`lvalue_expr.field`	`lvalue_expr++`
`expr->field`	`lvalue_expr`

Lvalue Expressions

Creation	Use
`variable`	`lvalue_expr = expr`
`*expr`	`&lvalue_expr`
`expr1[expr2]`	`lvalue_expr += expr`
`lvalue_expr.field`	`lvalue_expr++`
`expr->field`	`lvalue_expr`

Lvalue Conversion

Lvalue Expressions

Creation	Use
`variable`	`lvalue_expr = expr`
`*expr`	`&lvalue_expr`
`expr1[expr2]`	`lvalue_expr += expr`
`lvalue_expr.field`	`lvalue_expr++`
`expr->field`	`lvalue_expr`

Lvalue or Array Conversion

Converted Lvalue Example

```
ptr<int> p ;
```

```
*p = 3 ;
```

```
int i = *p ;
```

Converted Lvalue Example

```
ptr<int> p ;
*p = 3 ;
```


```
int i = *p ;
```

Converted Lvalue Example

```
ptr<int> p ;
*p = 3 ;
int i = *p ;
```

Value Expression bounds

```
expr : bounds(l, u)          ⊢ expr + i : bounds(l, u)

lvalue_expr : bounds(l, u)   ⊢ &lvalue_expr : bounds(l, u)
```

Lvalue Expression bounds

```
T var;                       ⊢ var : bounds(&var, &var + 1)

T arr checked[N];            ⊢ arr : bounds(arr, arr + N)

expr : bounds(l, u)          ⊢ *expr : bounds(l, u)
```

Lvalue *Target* bounds

```
array_ptr<T> var : bounds(l, u);   ⊢ var : bounds(l, u)

array_ptr<T> mem : bounds(l, u);   ⊢ x->mem : bounds(x->l, x->u)
```

Full Propagation Algorithm

In the Report

Description of Limitations

Dynamic Checks Are Performed During Evaluation of Lvalue Expressions that will Access Memory

When Do Dynamic Checks Occur?

```
int i = *p;
*p = 0;

*p += 1;

(*p)++;
```

Pointer Dereference

When Do Dynamic Checks Occur?

```
int i = *p;
```

Pointer Dereference → `p[n]`

Pointer Dereference → `p->field`

```
*p = 0;

*p += 1;

(*p)++;
```

When Do Dynamic Checks Occur?

```
int i = *p;
```

Pointer Dereference → `p[n]`, `p->field`

```
*p = 0;
```

Assignment

```
*p += 1;
```

Compound Assignment

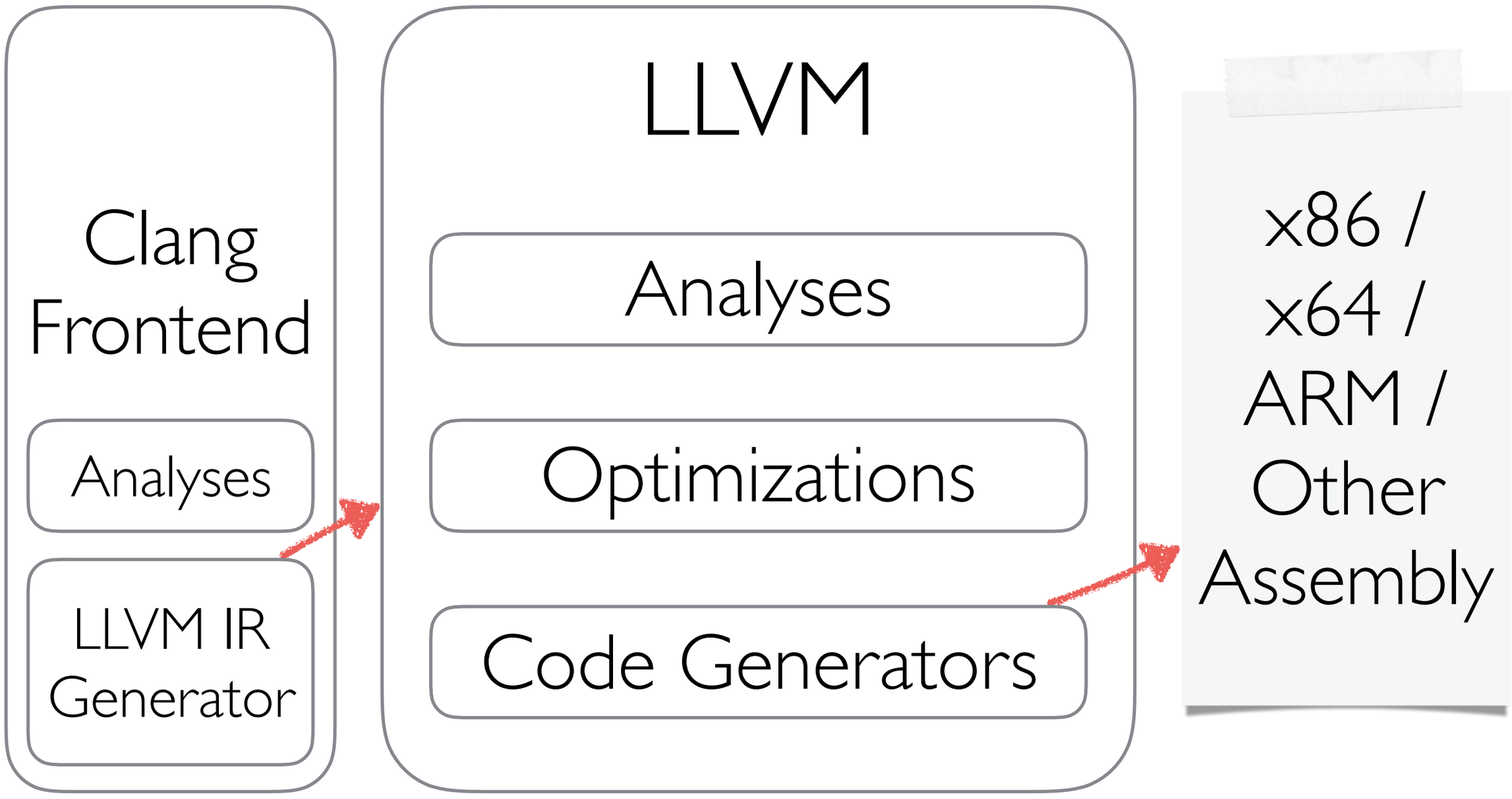
```
(*p)++;
```

Increment/Decrement

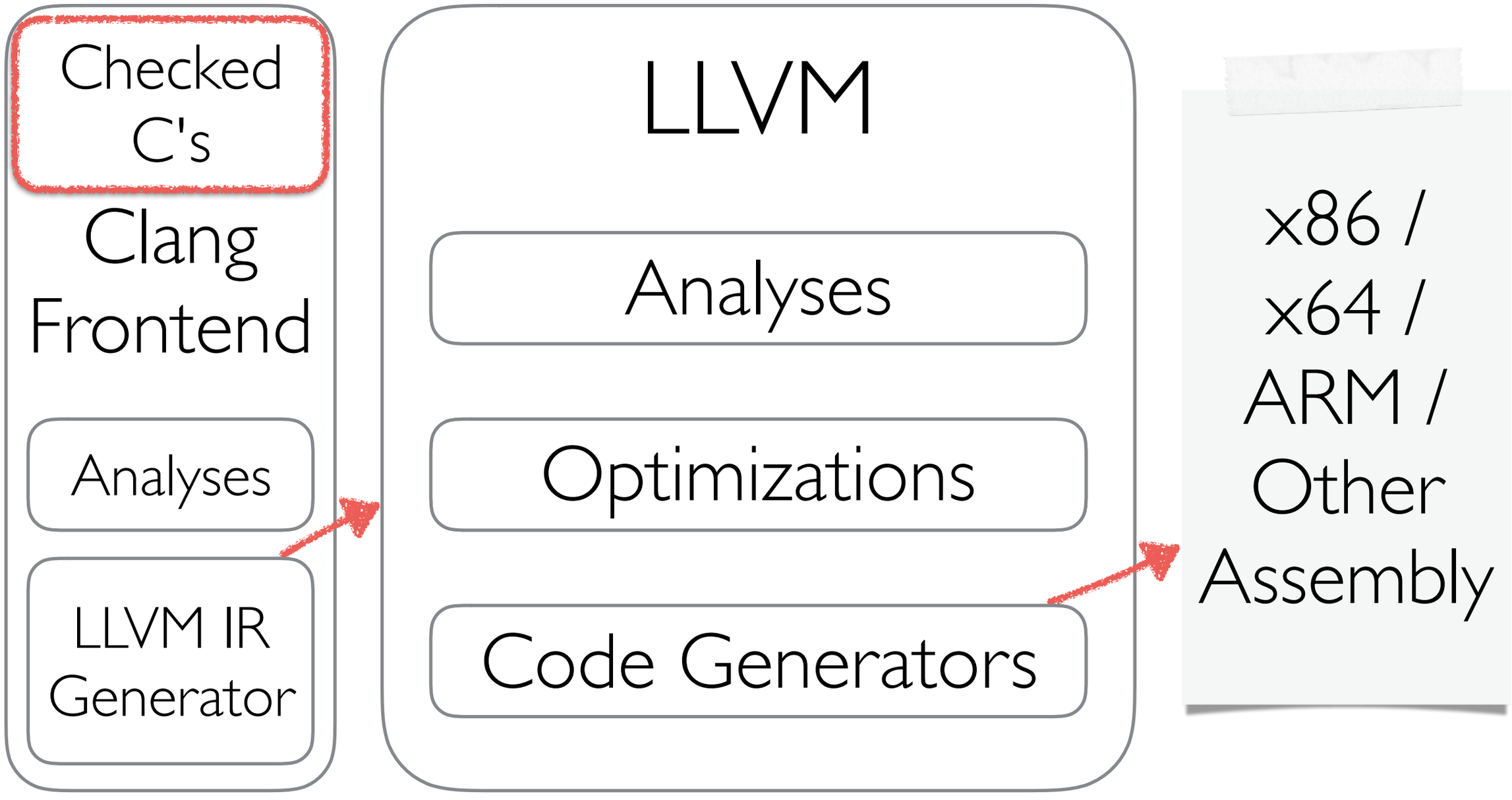
Design Requirements

- No Runtime Library
- Cross-Platform
- No Implicit Overhead except where Memory is Accessed
    - So no checks in calls, no checks in pointer arithmetic

Clang & LLVM


Clang Frontend
Analyses
LLVM IR Generator
LLVM
Analyses
Optimizations
Code Generators
x86 / x64 / ARM / Other Assembly

Clang & LLVM


Checked C's
Clang Frontend
Analyses
LLVM IR Generator
LLVM
Analyses
Optimizations
Code Generators
x86 / x64 / ARM / Other Assembly

Clang's Code Generator

Clang's Code Generator

- Value Expressions:
  - Scalar Expressions
  - Vector Expressions
  - Aggregate Types
  - Complex Numbers
- Lvalue Expressions

Clang's Code Generator

- Value Expressions:
  - Scalar Expressions
  - Vector Expressions
  - Aggregate Types
  - Complex Numbers
- Lvalue Expressions

All Generate LLVM IR

Clang's Code Generator

- Value Expressions:
  - Scalar Expressions
  - Vector Expressions
  - Aggregate Types
  - Complex Numbers
- Lvalue Expressions

All Generate LLVM IR

LLVM IR is in SSA form, similar to Assembly, but platform independent

Clang's Code Generator

- Value Expressions:
  - Scalar Expressions
  - Vector Expressions
  - Aggregate Types
  - Complex Numbers
- Lvalue Expressions

All Generate LLVM IR

LLVM IR is in SSA form, similar to Assembly, but platform independent

Dynamic Checks Generated Here

- Checked C
- Example
- Dynamic Checks

- Checked C
- Example
- Dynamic Checks
- **Evaluation**

Hypotheses

- Most `T*` become `ptr<T>`

- 10% of Code Changed

- 10-50% Slower Run-time

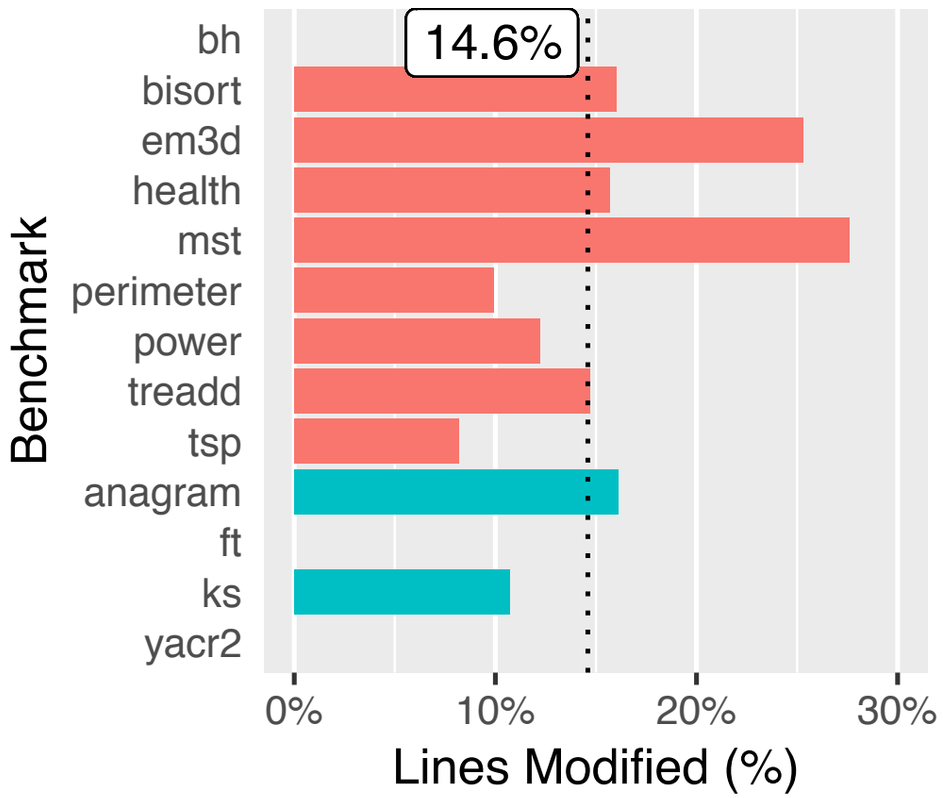
Benchmarks

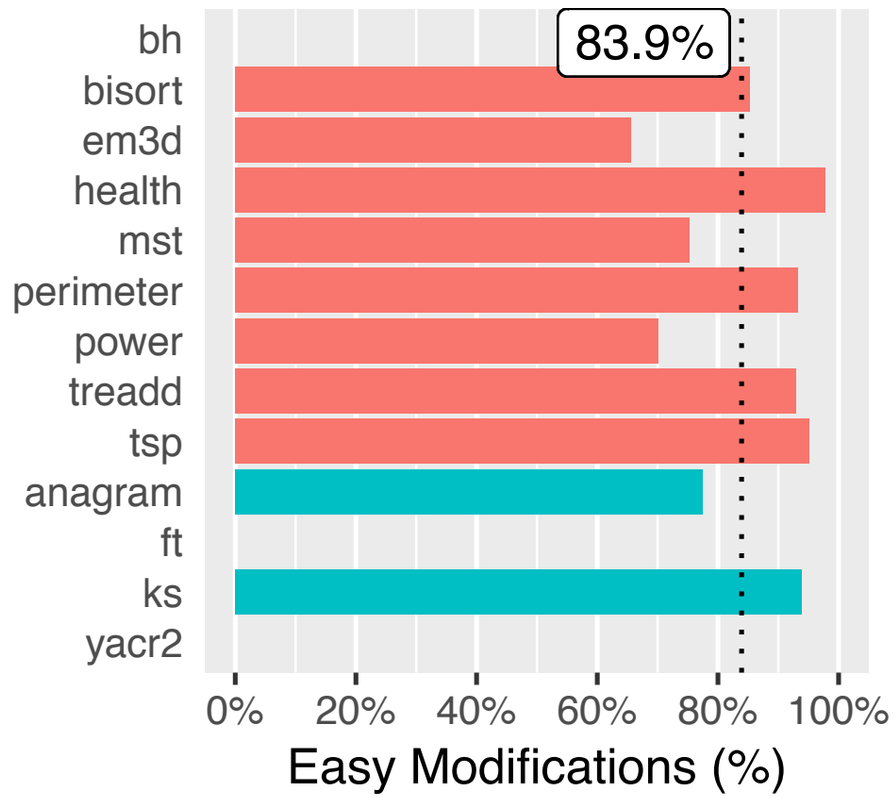
- Olden and Ptrdist Suites: 15 Programs
- Converted By Hand
  - 5 with assistance from Wonsub Kim and Jijoong Moon at Samsung Research
  - 2 Conversions Incomplete
- 12 Core Intel Xeon X5650, 2.66GHz, 32GB RAM

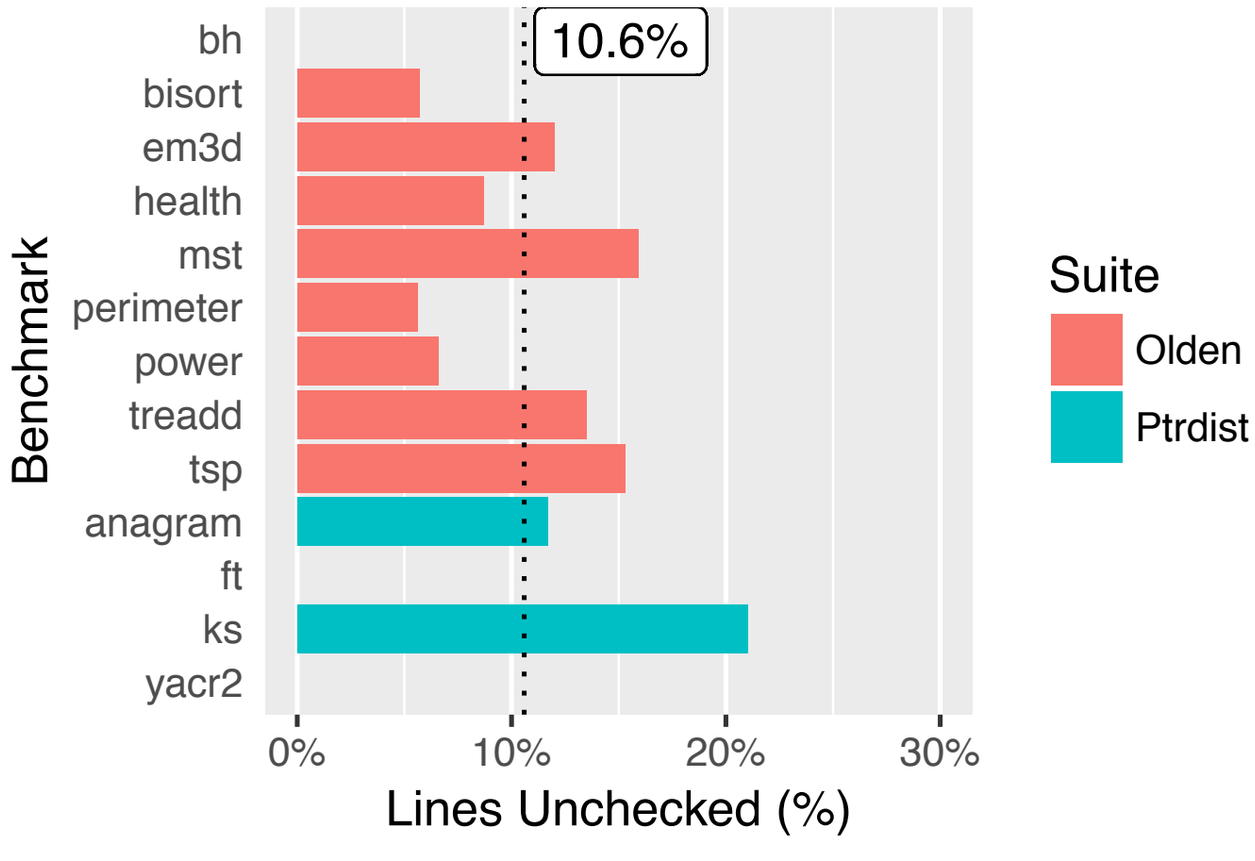
Benchmarks

Benchmark	LoC	Description
Olden: bh	1,162	Barnes & Hut N-body force computation
Olden: bisort	263	Forward & Backward Bitonic Sort
Olden: em3d	478	3D Electromagnetic Wave Propagation
Olden: health	389	Columbian Health Care Simulation
Olden: mst	328	Minimum Spanning Tree
Olden: perimeter	399	Perimeters of Regions on Images
Olden: power	458	Power Pricing Optimisation Solver
Olden: treeadd	180	Recursive Sum over Tree
Olden: tsp	420	Travelling Salesman Problem
Olden: voronoi	814	Computes voronoi diagram of a set of points
Ptrdist: anagram	362	Finding Anagrams from a Dictionary
Ptrdist: bc	5,194	Arbitrary precision calculator
Ptrdist: ft	893	Minimum Spanning Tree using Fibonacci heaps
Ptrdist: ks	552	Schweikert-Kernighan Graph Partitioning
Ptrdist: yacr2	2,529	VSLI Channel Router

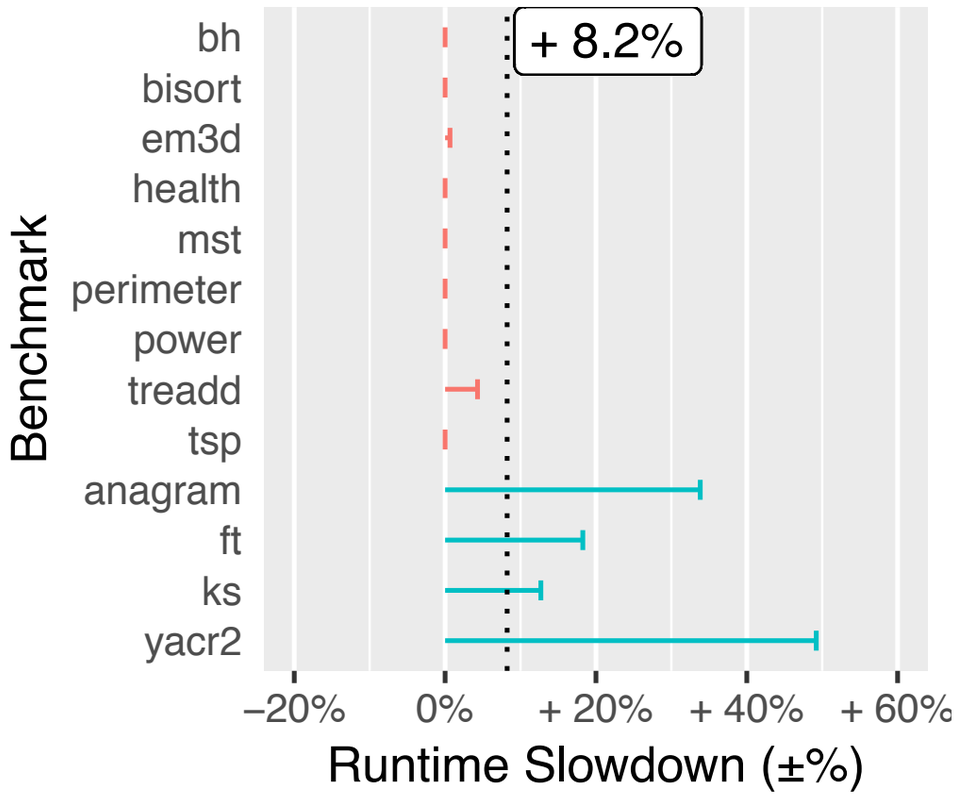
Code Modifications

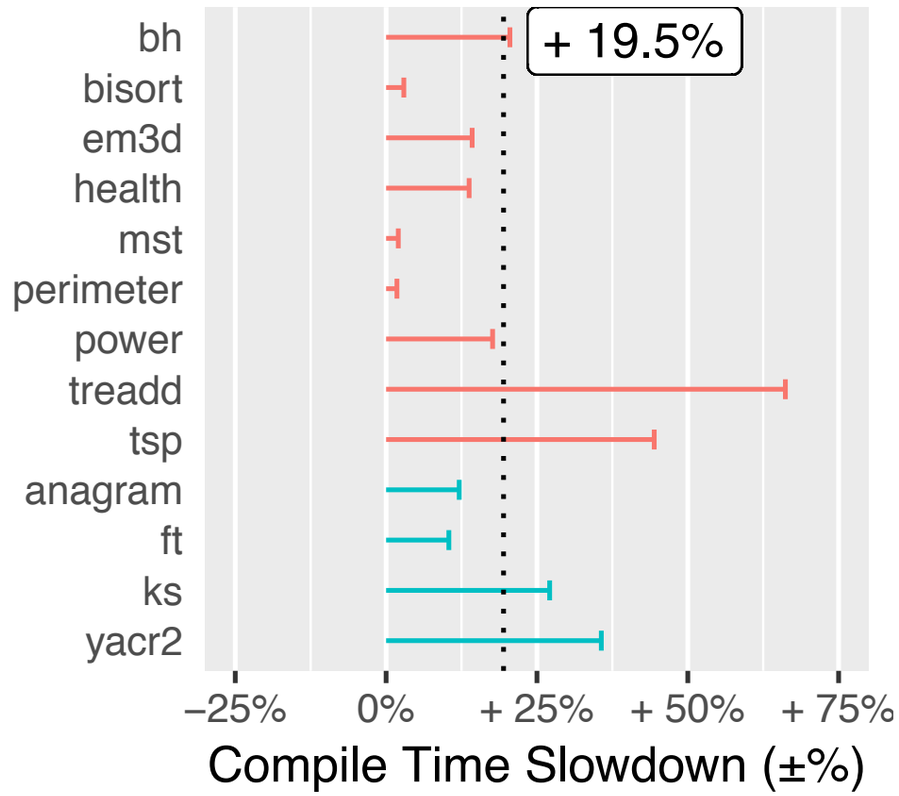


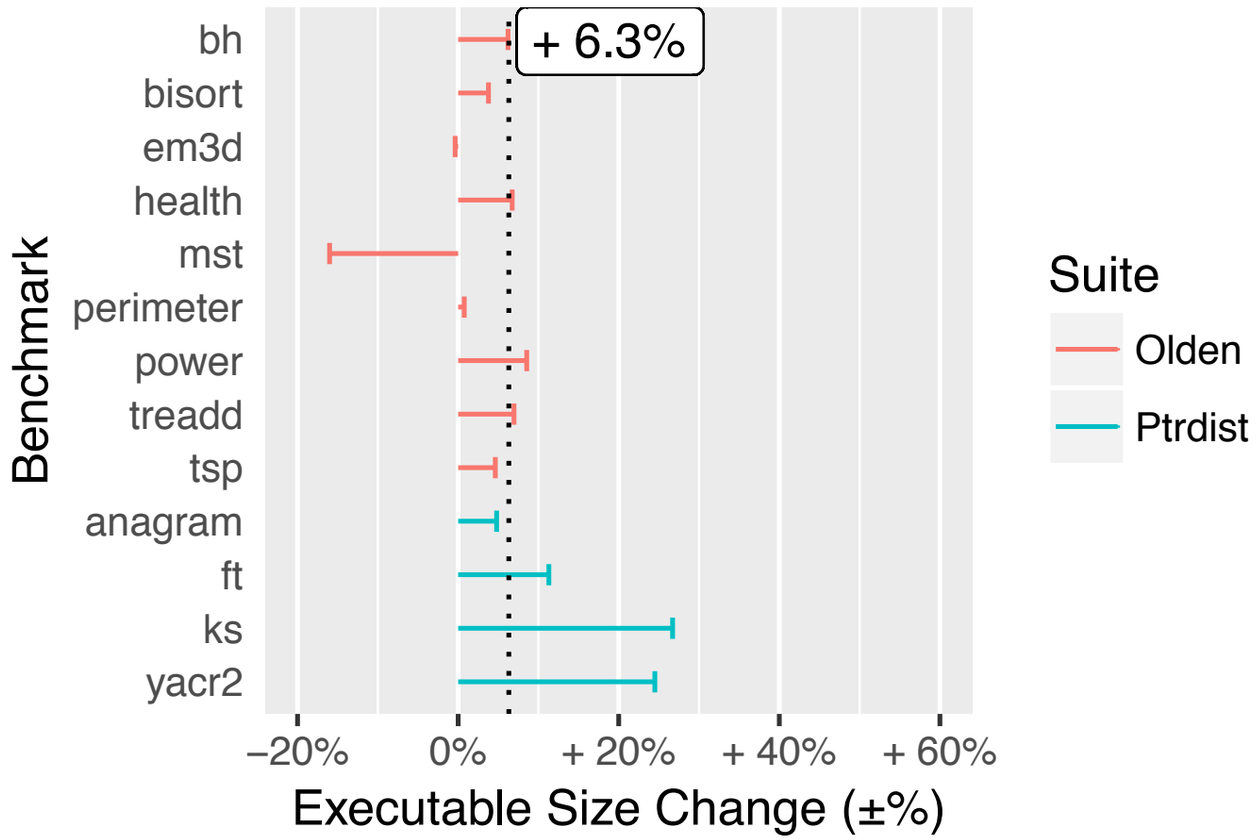




Performance Overhead







LLVM Optimizer

~250 Analyses and Optimizations

Most Useful:

- CSE/GVN
- InstCombine
- Simplify CFG
- LICM
- Loop Unswitching

LLVM Optimizer

~250 Analyses and Optimizations

Most Useful:

- CSE/GVN
- InstCombine
- Simplify CFG
- LICM
- Loop Unswitching

Major Problems:

- Writing Good Bounds
- Non-null Checks

- Checked C
- Example
- Dynamic Checks
- **Evaluation**

- Checked C
- Example
- Dynamic Checks
- Evaluation
- **Conclusion**

C Extension for Spatial Memory Safety

Bounds Propagation Algorithm

Added Runtime **Bounds Checks** to the **Checked C** Compiler

First Evaluation of Checked C

In Our Clang/LLVM Fork

Next Steps

- Design for Null-Terminated Arrays
- Better Static Checking of Bounds
- Proposal for nullary qualifiers to reduce overhead
- Evaluating on Real-world Benchmarks

Working on a Paper with Andrew Ruef, Michael Hicks, and David Tarditi

C Extension for Spatial Memory Safety

Bounds Propagation Algorithm

Added Runtime **Bounds Checks** to the **Checked C** Compiler

First Evaluation of Checked C

In Our Clang/LLVM Fork

https://github.com/Microsoft/checkedc
https://github.com/Microsoft/checkedc-clang