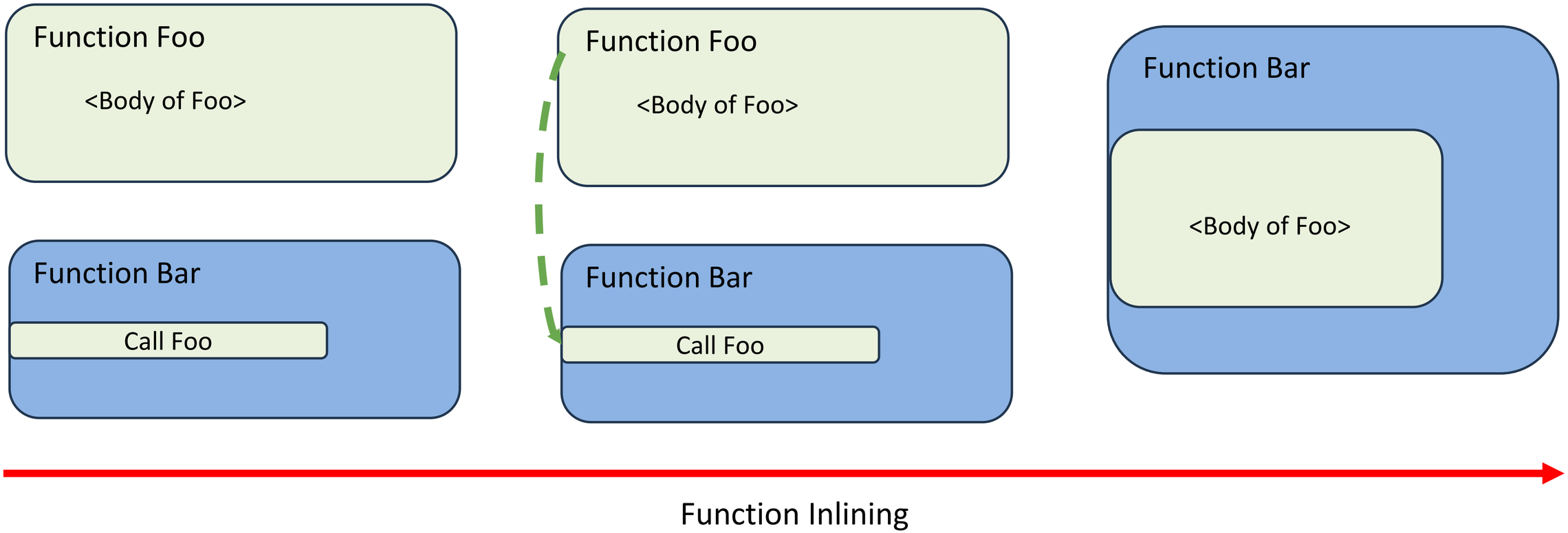
A Deep Dive into Function Inlining and its Security Implications for ML-based Binary Analysis

Omar Abusabha, Jiyong Uhm, Tamer Abuhmed, and Hyungjoon Koo

Sungkyunkwan University

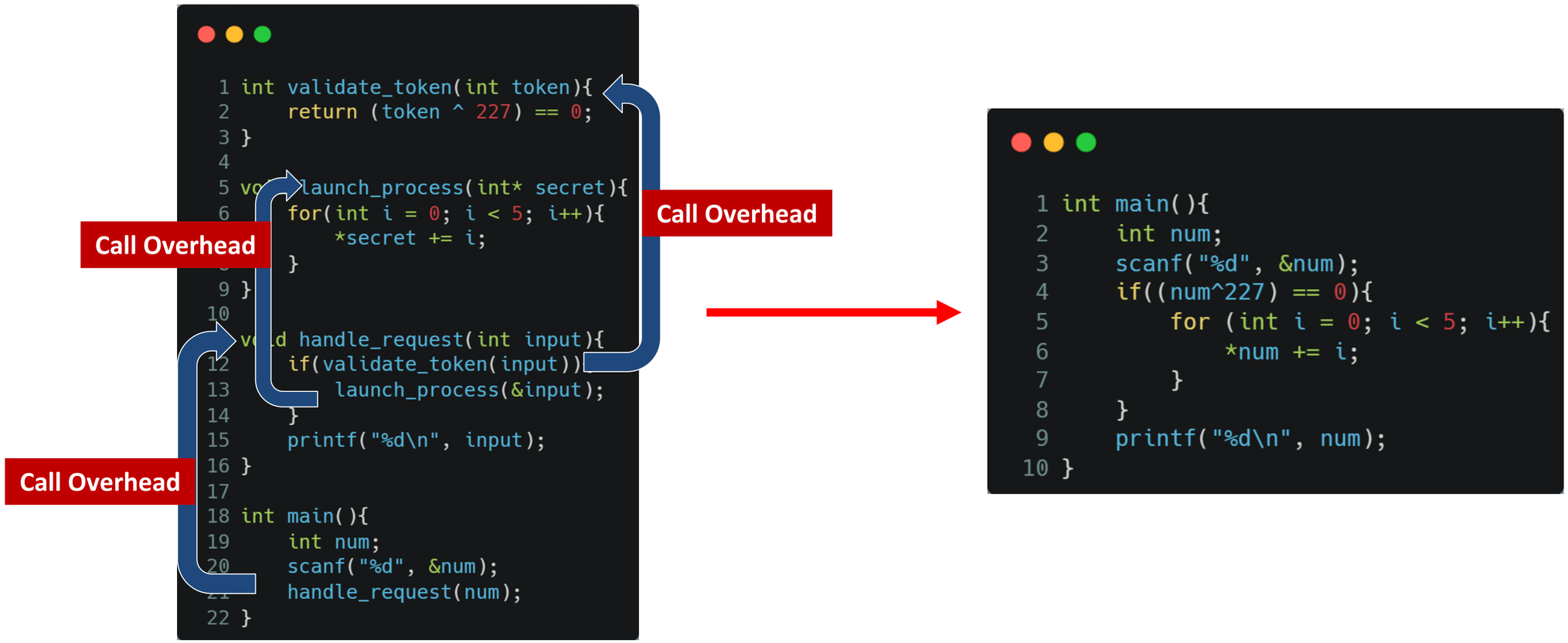
Function Inlining

- One of the common optimization techniques in a modern compiler
- Replaces a call site with the callee's body



Function Inlining Benefits

- Decreases a function call overhead and increases cache locality for performance
- Enables further optimizations after inlining



Function Inlining Impacts

```
<main>:
push           rax
lea            rdi,[rip+0xe50]
lea            rsi,[rsp+0x4]
xor            eax,eax
call           1040 <__isoc99_scanf@plt>
mov            edi,DWORD PTR [rsp+0x4]
call           1170 <handle_request>
xor            eax,eax
pop            rcx
ret
```

Without inlining

Static features changes

- Control flow graph
- Call graph
- Instructions

```
<main>:
push   rax
lea    rdi,[rip+0xe30]
lea    rsi,[rsp+0x4]
xor    eax,eax
call   1040 <__isoc99_scanf@plt>
mov    esi,DWORD PTR [rsp+0x4]
cmp    esi,0xe3
jne    1213 <main+0x43>
cmp    DWORD PTR [rip+0x2e29],0x0
jle    120e <main+0x3e>
xor    eax,eax
mov    esi,0xe3
add    esi,eax
inc    eax
cmp    eax,DWORD PTR [rip+0x2e16]
jl     1200 <main+0x30>
jmp    1213 <main+0x43>
mov    esi,0xe3
lea    rdi,[rip+0xdea]
xor    eax,eax
call   1030 <printf@plt>
xor    eax,eax
pop    rcx
ret
```

With inlining

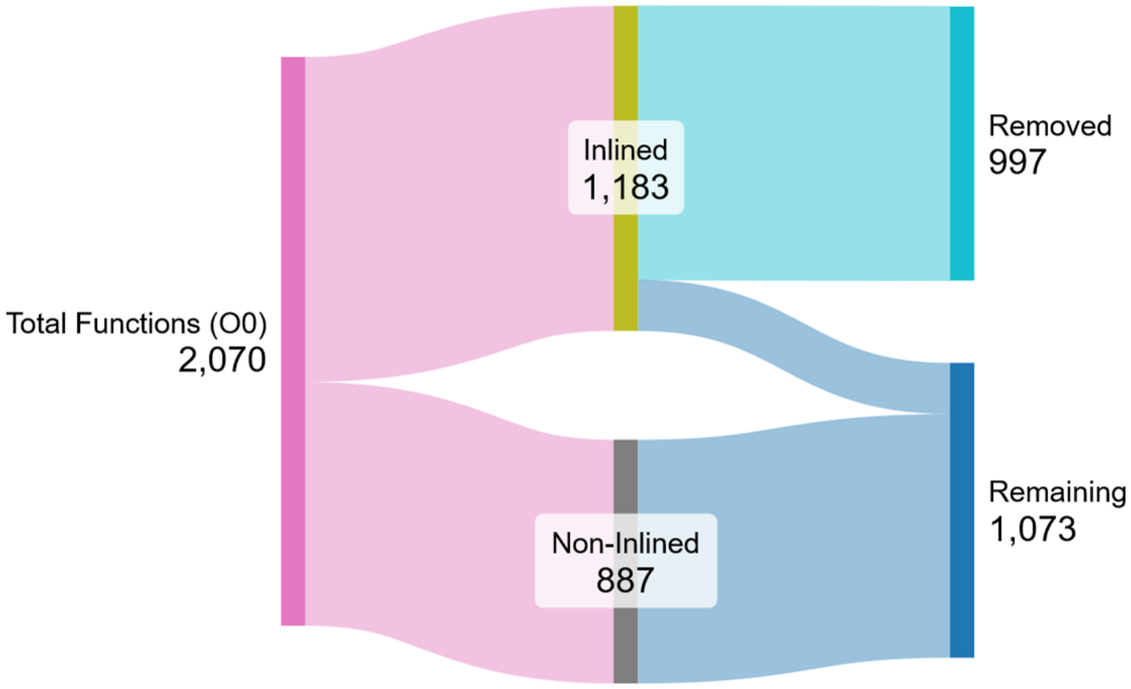
ML-based Models (often) Rely on Static Features

- Static Features (BinKit, TSE ’22)

Instruction-level features	Control Flow Graph (CFG)	Call Graph (CG)
(Average) Number of instructions	Size	Number of callers/callees
(Average) Number of unknown instructions	(Average) Number of loops	Number of imported callees
(Average) Number of absolute arithmetic instructions	(Average) Number of interprocedural loops	Number of incoming calls
(Average) Number of arithmetic instructions	(Average) Number of strongly connected components	Number of outgoing calls
(Average) Number of comparison instructions	Number of back edges	Number of imported calls
(Average) Number of absolute control transfer instructions	Number of breadth-first search edges	…
(Average) Number of conditional control transfer instructions	Maximum width/depth	
(Average) Number of group jump instructions	Sum of the sizes of all loops	
(Average) Number of absolute data transfer instructions	Sum of the sizes of all interprocedural loops	
(Average) Number of data transfer instructions	Sum of the sizes of all strongly connected components	
(Average) Number of control transfer instructions	(Average) Number of incoming/outgoing edges	
(Average) Number of group call instructions	(Average) Number of edges (in-degree + out-degree)	
(Average) Number of group return instructions	…	
…		

Function Inlining Prevalence

- Common transformation, disappearing inlined functions
- Affects static features significantly



Coreutils (-O3 optimization)

Motivations

- How robust are ML-based security models to inlining-induced feature drift?
- Can adversaries leverage an inlining rate to evade or mislead these models?

Roadmap

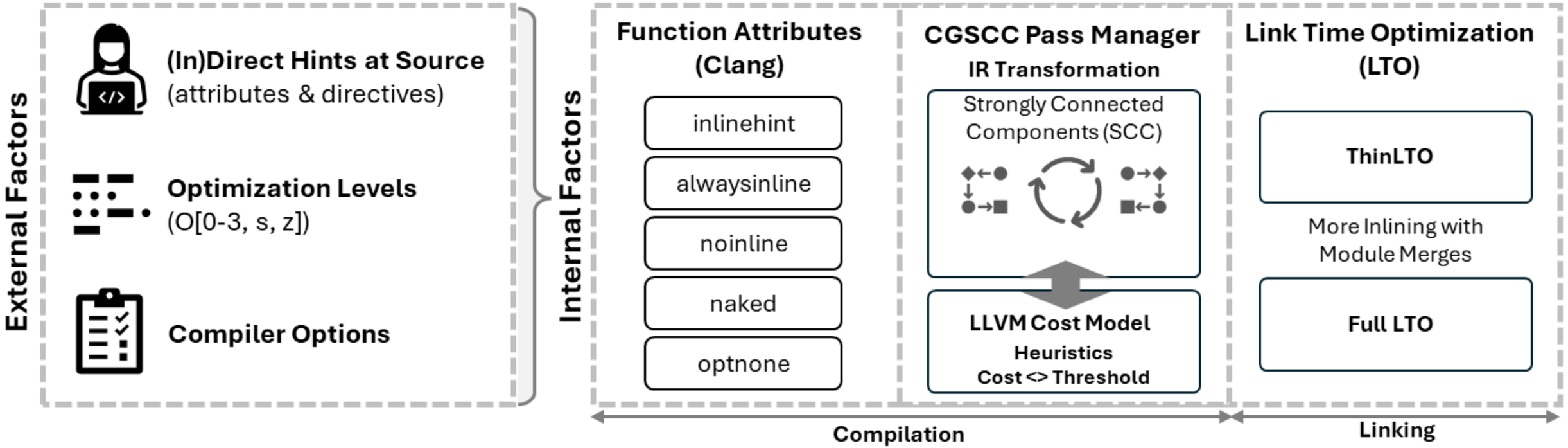
- Investigates the function inlining optimization in the LLVM Compiler Infrastructure
- Explores how inlining behavior can be controlled (extreme inlining)
- Evaluates model performance against function inlining and extreme inlining

Function Inlining Pipeline (LLVM)

External Factors
(Source Code, Compiler Flags)

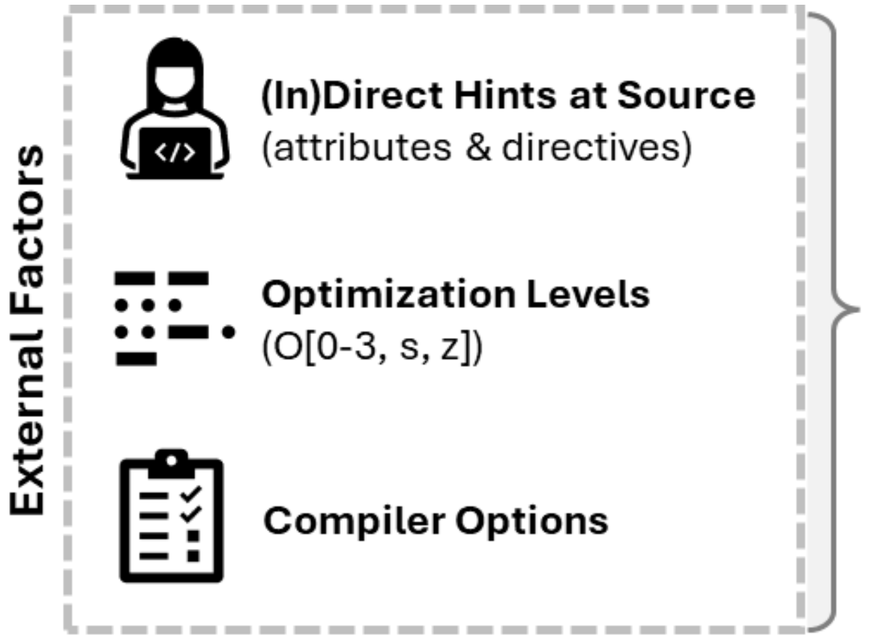
Internal Factors
(Compiler, Linker)

Function Inlining Pipeline (LLVM)



External Factors that Determine Inlining

- Attributes and directives
  e.g., *always_inline*
- Optimization levels
  e.g., O1, O2, O3
- Flags
  e.g., *-fno-inline*, *-inline-threshold*

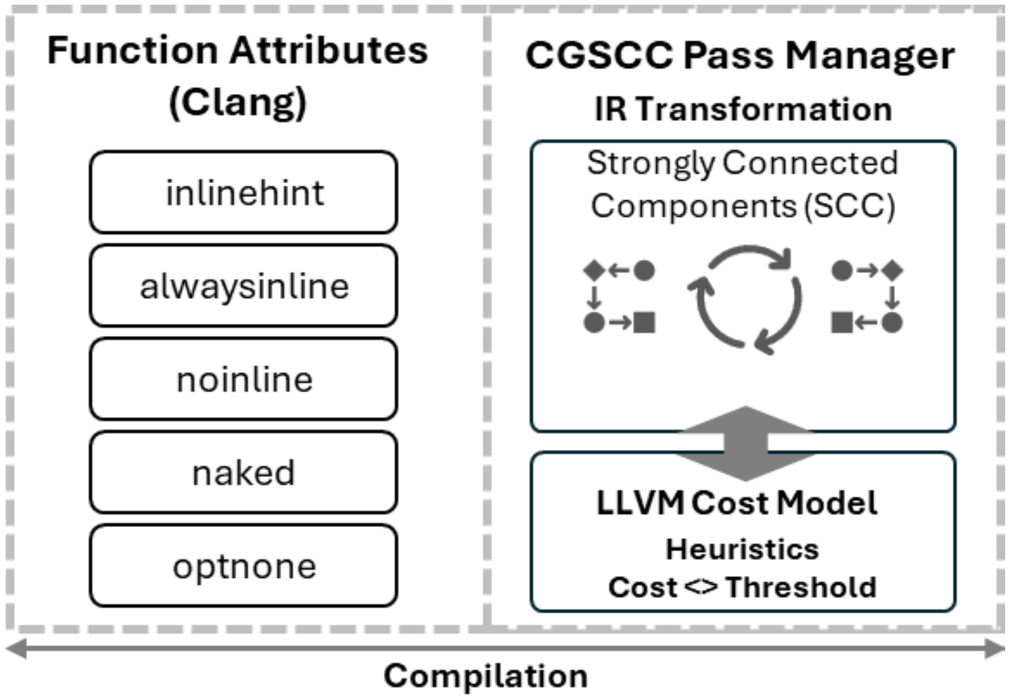


Internal Factors that Determine Inlining

- Function Attributes
  e.g., *alwaysinline, noinline*

- Built-in Inlining Heuristics
  e.g., Always or Never Inline

- Inlining Cost Model

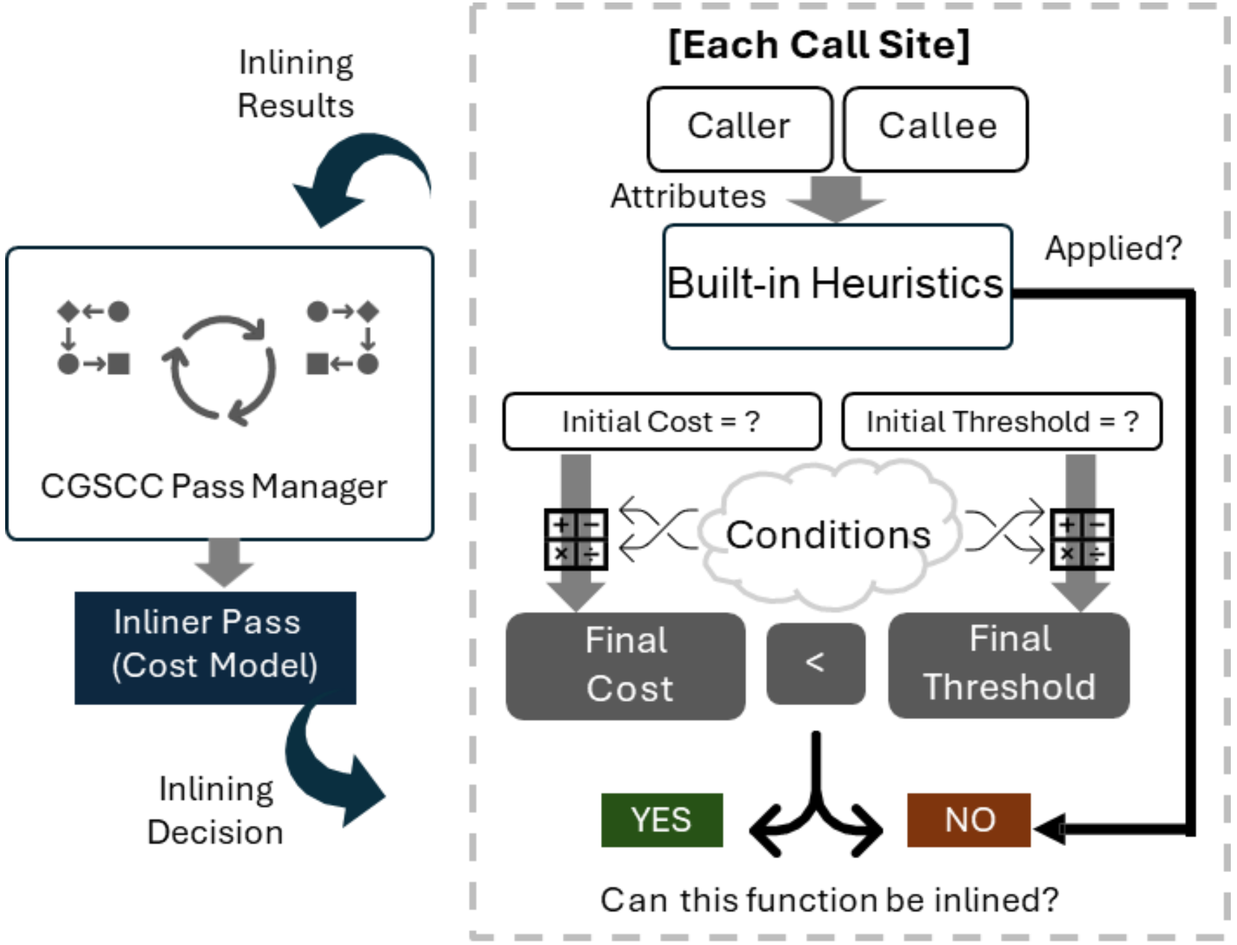


Built-in Inlining Heuristics

- Inlining never happens when:
  - Function has certain attributes
    e.g, *optnone*, *noinline*
  - Function Characteristics
    e.g., recursive or interposable

Cases where Inliner Pass will not inline	
Caller	*optnone* attribute (disable optimization)
Callee	*noinline* attribute (disable inlining)
	Recursive function
	Interposable function (replaced/overridden at link time or runtime)

Function Inlining Cost Model in LLVM



What if an attacker controls inlining behavior by leveraging the cost model? (i.e., staying compiler internals intact)

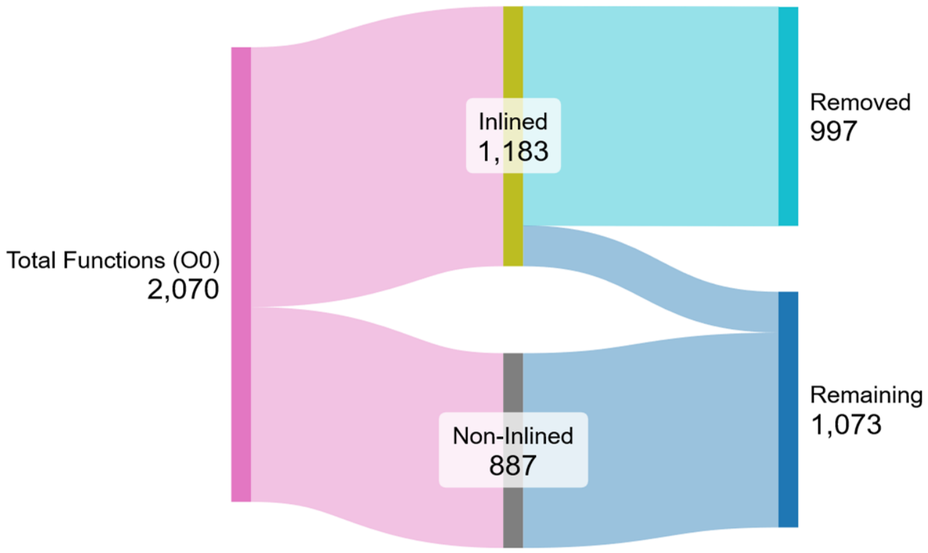
Extreme Inlining

- Exploiting compiler flags (options) only
  - No change in compiler internals
  - Recipe for increasing the inlining rate toward *extreme inlining*

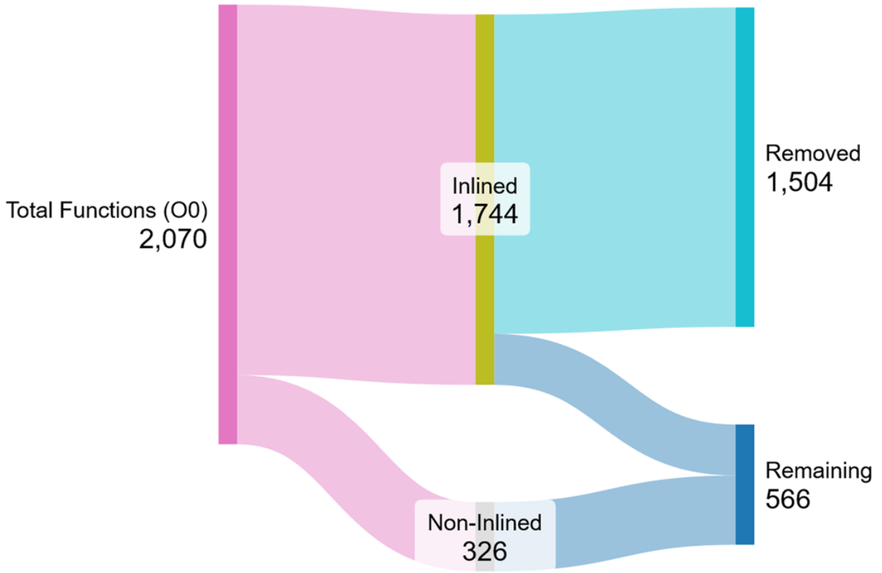
- Notable examples:
  - *-inline-threshold:* alters the initial threshold in the cost model
  - *-flto=full*: enables link time optimization

Extreme Inlining Example

- Compiler flags that can aggressively increase the inlining ratio
  - *e.g., -O3 –flto=full –inline-threshold=200000*



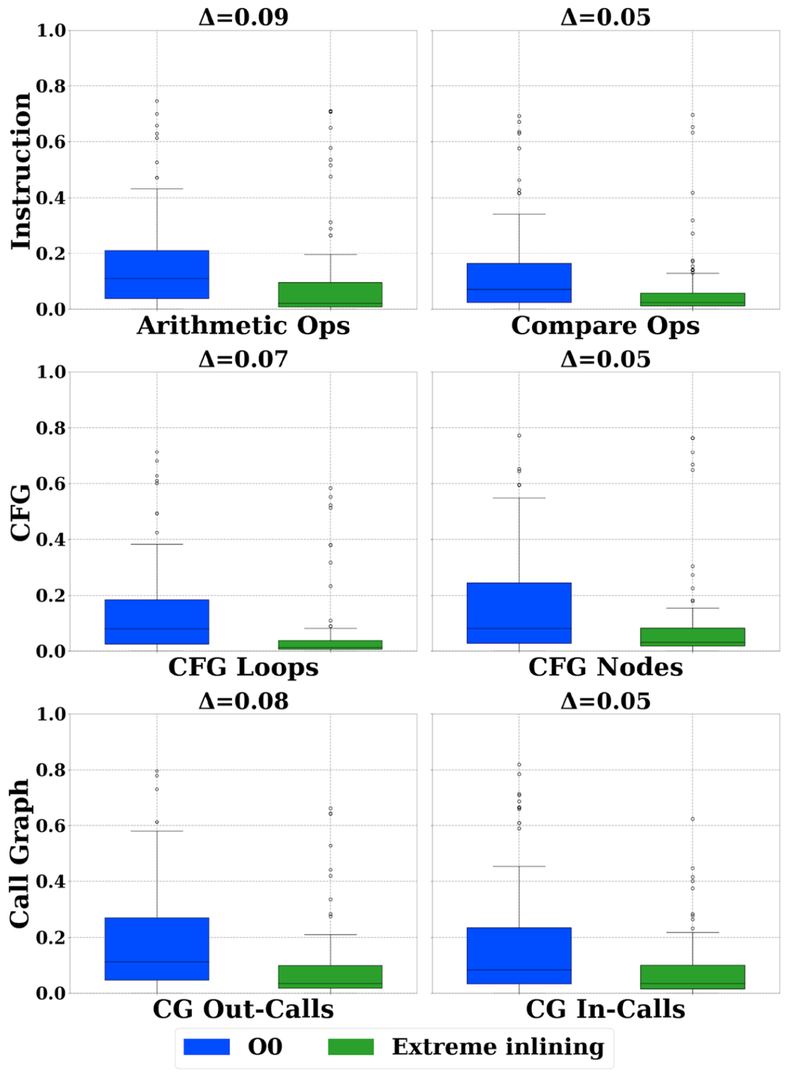
O3 Optimization



Extreme Inlining

Static Feature Changes with Extreme Inlining

- Extreme inlining alters static features
  - Normalized values of the corresponding static features
    e.g., # of arithmetic operations
  - Changes across various static features



Δ represents difference in the mean

What is the impact of extreme inlining on ML models?

Impact of Function Inlining for ML models

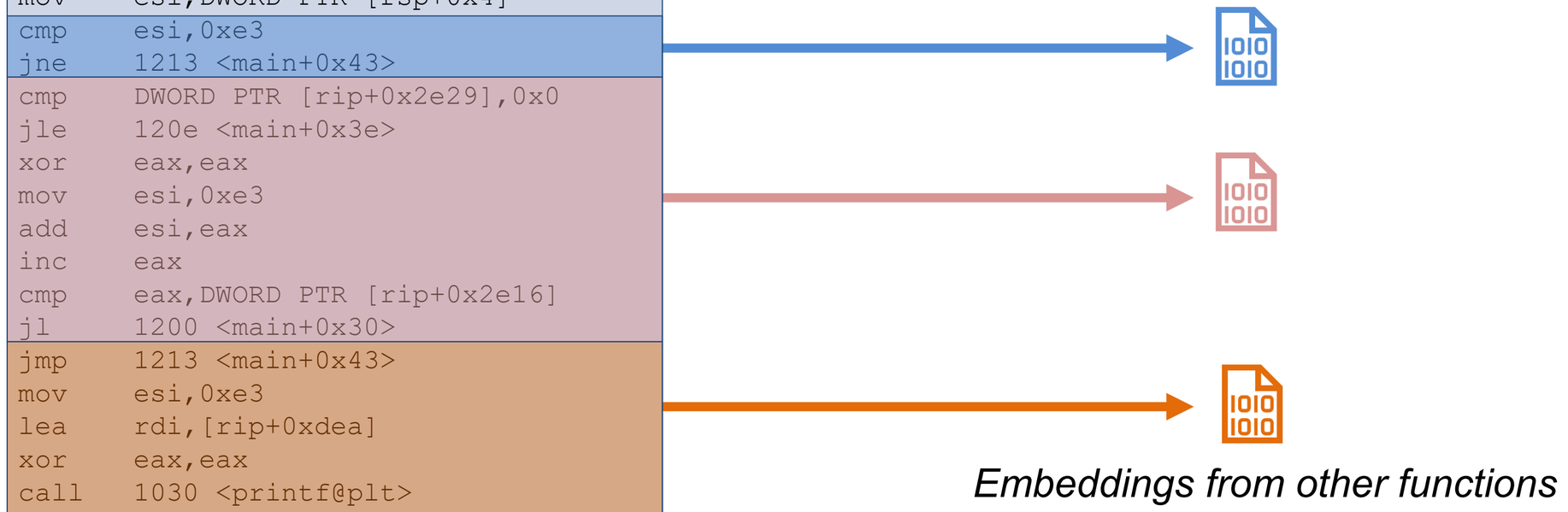
- Multiple contexts present within one function

```c
int validate_token(int token){
    return (token ^ 227) == 0;
}

void launch_process(int* secret){
    for(int i = 0; i < 5; i++){
        *secret += i;
    }
}

void handle_request(int input){
    if(validate_token(input)){
        launch_process(&input);
    }
    printf("%d\n", input);
}

int main(){
    int num;
    scanf("%d", &num);
    handle_request(num);
}
```

```
<main>:
push   rax
lea    rdi,[rip+0xe30]
lea    rsi,[rsp+0x4]
xor    eax,eax
call   1040 <__isoc99_scanf@plt>
mov    esi,DWORD PTR [rsp+0x4]
cmp    esi,0xe3
jne    1213 <main+0x43>
cmp    DWORD PTR [rip+0x2e29],0x0
jle    120e <main+0x3e>
xor    eax,eax
mov    esi,0xe3
add    esi,eax
inc    eax
cmp    eax,DWORD PTR [rip+0x2e16]
jl     1200 <main+0x30>
jmp    1213 <main+0x43>
mov    esi,0xe3
lea    rdi,[rip+0xdea]
xor    eax,eax
call   1030 <printf@plt>
xor    eax,eax
pop    rcx
ret
```

Machine Learning based Binary Analysis

```
<main>:
push   rax
lea    rdi,[rip+0xe30]
lea    rsi,[rsp+0x4]
xor    eax,eax
call   1040 <__isoc99_scanf@plt>
mov    esi,DWORD PTR [rsp+0x4]
cmp    esi,0xe3
jne    1213 <main+0x43>
cmp    DWORD PTR [rip+0x2e29],0x0
jle    120e <main+0x3e>
xor    eax,eax
mov    esi,0xe3
add    esi,eax
inc    eax
cmp    eax,DWORD PTR [rip+0x2e16]
jl     1200 <main+0x30>
jmp    1213 <main+0x43>
mov    esi,0xe3
lea    rdi,[rip+0xdea]
xor    eax,eax
call   1030 <printf@plt>
xor    eax,eax
pop    rcx
ret
```



Embeddings from other functions

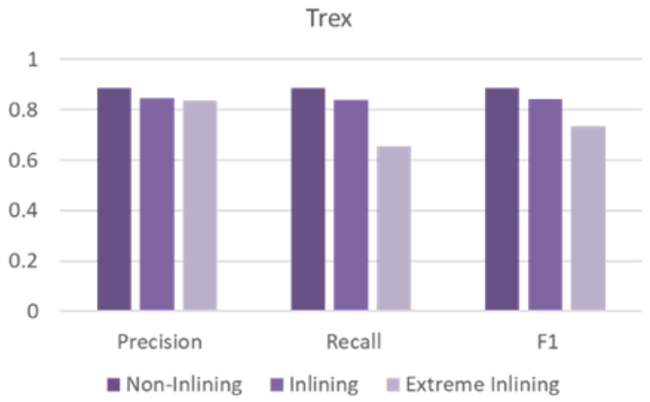
Impact of Function Inlining on ML-based Security Tasks

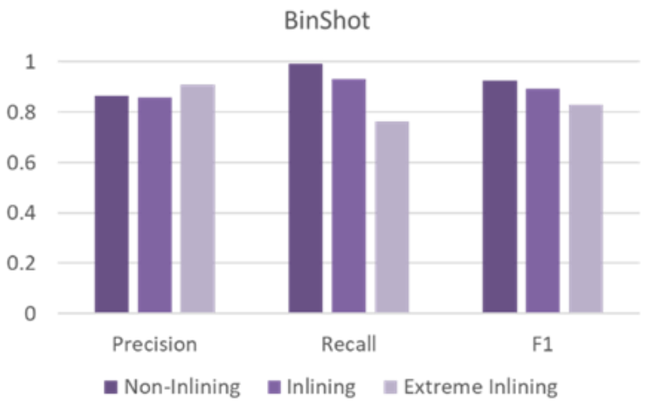
- (T1) Binary Code Similarity Detection
- (T2) Function Name Inference
- (T3) Malware Detection and Malware Family Prediction
- (T4) Vulnerability Detection

(T1) Binary Code Similarity Detection (BCSD)

- (Drop in recall) Similar functions appear dissimilar to the model after inlining

- Additional semantics from other functions
- Additional optimization due to inlining

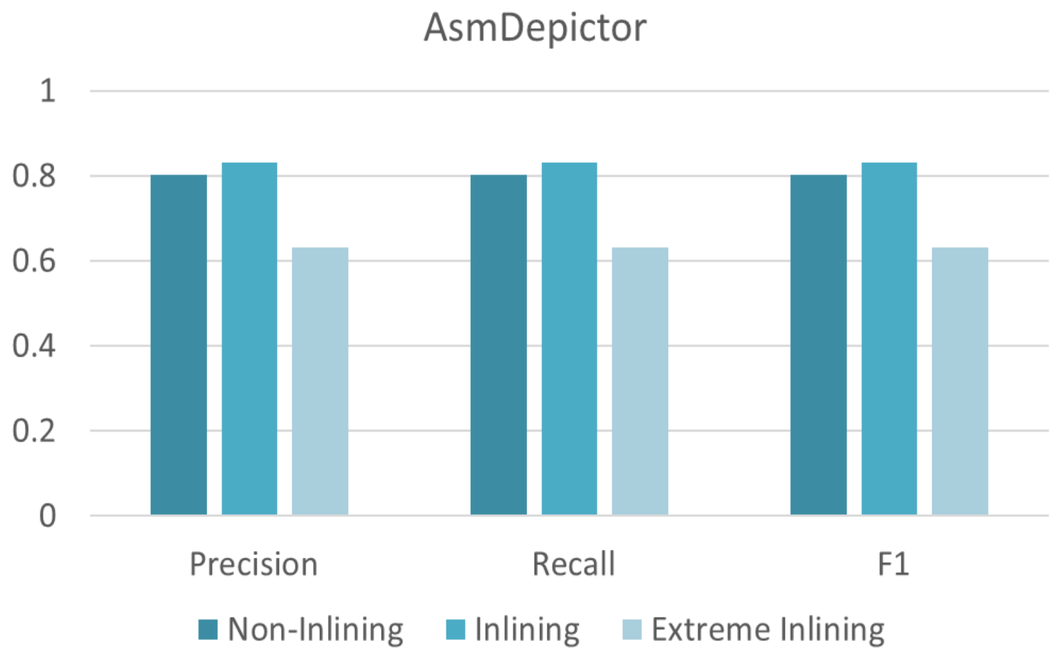




(T2) Function Name Prediction

- Inlining improves performance (leveraging additional semantic information)

- Extreme inlining introduces abnormal inlining patterns



(T3) Malware Detection and Family Prediction

- Substantial changes in code structure ⇒ Misclassification
- Broader impact on malware family prediction
  - Extreme inlining obscures structural patterns essential for prediction

Security Task	Malware in the Wild				Malware with Extreme Inlining			
	Accuracy	Precision	Recall	F1	Accuracy	Precision	Recall	F1
Malware Detection	0.98 ± 0.01	0.98 ± 0.01	0.98 ± 0.01	0.98 ± 0.01	0.77 ± 0.05	0.78 ± 0.04	0.79 ± 0.01	0.78 ± 0.05
Malware Family Prediction	0.87 ± 0.05	0.88 ± 0.05	0.87 ± 0.05	0.87 ± 0.05	0.44 ± 0.09	0.46 ± 0.09	0.43 ± 0.09	0.41 ± 0.10

(T4) Vulnerability Detection

- Average drop of 69% after inlining
- Trex on Netgear R7000
  - One vulnerable function had a higher rank while remaining had a lower rank

Model	Netgear R7000		TP-Link Deco-M4	
	Non-Inlining	Inlining	Non-Inlining	Inlining
Gemini*	0.049	0.000	0.321	0.004
Asm2Vec	0.256	0.128	0.016	0.008
SAFE	0.125	0.003	0.018	0.003
Trex	0.550	0.625	0.286	0.048

MRR @ 100: Mean Reciprocal Rank 100 (Larger the Better)

**Not Google's Gemini*

Limitations

- Lack of representativeness in our dataset
- Inlining behavior differences across different compilers
  - GCC briefly explored (details in the paper)
- Limited metric for an inlining ratio
  - Does not capture consecutive inlining into the same function

Leftover – See Our Paper!

- Inlining investigation
  - Static feature changes under inlining
  - Inlining under link time optimization (LTO)

- Exploration of how inlining can be controlled
  - Inlining ratio across applications and optimization levels
  - Impact of compiler options on inlining ratio
  - Extreme inlining exploration

- Mitigations
  - Training data augmentation with extreme inlining

Takeaways

- Function inlining of ML-based models for binary analysis
  - Causes non-negligible static feature changes
  - Could be leveraged by an attacker

- Robust models require
  - Accounting for inlining-induced feature changes
  - Considering inlining-aware processing

Thank you, any questions?

https://doi.org/10.5281/zenodo.17759528

404970@g.skku.edu

Common Misconceptions About Function Inlining

- An inlined function would disappear in a binary

External Linkage

```c
// External Linkage
int add(int a, int b){
    return a + b;
}

int main(){
    return a(1,2);
}

```

Non-Inlined Callsite Remains

```c
static int square(int x){
    return x*x;
}

// Disables optimization for this function
// square() is not inlined!
__attribute__((optnone))  int two_squares(int x){
    int s = square(x);
    return s * 2;
}

int main(){
    int s = square(10);
    printf("Square : %d", s);
    int s2 = two_squares(10);
    printf("Two : %d", s2);
}
```

Common Misconceptions About Function Inlining

- Options that force or disable inlining (e.g., `*-fno-inline*`, `*always-inline*`)

Always Inline attribute overrides *fno-inline* and *-O0*

```
1 __attribute__((always_inline))
2 int add(int a, int b){
3     return a + b;
4 }
```

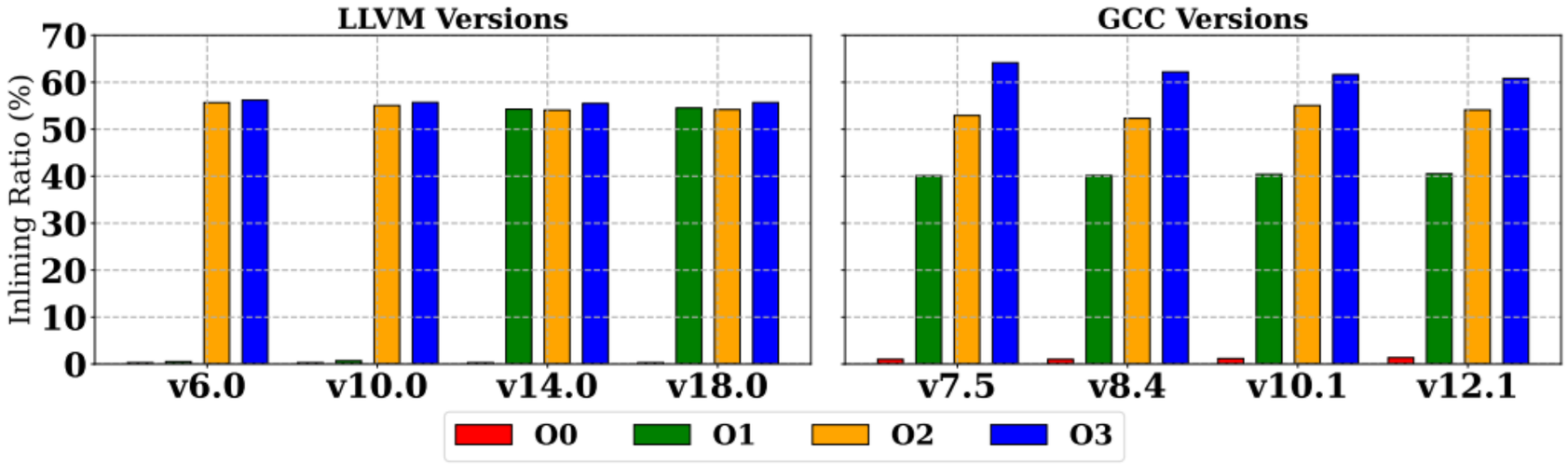
Some functions are never inlined (e.g., recursive functions*)

```
1 __attribute__((always_inline)) static void mergesort(int *a, int l, int r) {
2     if (l >= r) return;
3     int m = (l + r) / 2;
4     mergesort(a, l, m);
5     mergesort(a, m + 1, r);
6     merge(a, l, m, r);
7 }
```

* In some cases, the compiler may turn a recursive function into a non recursive function and apply inlining

Across Compilers

- Optimization level does control the inlining ratio
- Inlining consistently still occur in O0 regardless (by a small amount)



Inlining Related Flags

Option Name	Tool	Default	Description
`-finline-functions`	Clang	Disabled	Inlines a suitable function based on an optimization level
`-finline-hint-functions`	Clang	Disabled	Inlines a function that is (explicitly or implicitly) marked as `inline`
`-fno-inline-functions`	Clang	Disabled	Disables function inlining unless a function is declared with `always_inline`
`-fno-inline`	Clang	Disabled	Disables all function inlining except `always_inline`
`-inlinedefault-threshold`	Opt	225	Sets the initial threshold for O1 and O2 alone
`-inline-threshold`	Opt	225	Sets the initial threshold for all optimization levels
`-inlinehint-threshold`	Opt	335	Sets the threshold of a function marked with the `inlinehint` attribute
`-inline-cold-callsite-threshold`	Opt	45	Sets the threshold of a cold call site
`-inline-savings-multiplier`	Opt	8	Sets the multiplier for cycle savings during inlining
`-inline-size-allowance`	Opt	100	Sets the size allowance for inlining without sufficient cycle savings
`-inlinecold-threshold`	Opt	45	Sets the threshold for a cold call site
`-hot-callsite-threshold`	Opt	3,000	Sets the threshold for a hot call site
`-locally-hot-callsite-threshold`	Opt	525	Sets the threshold for a hot call site within a local scope
`-cold-callsite-rel-freq`	Opt	2	Sets a maximum block frequency for a call site to be cold (no profile information)
`-hot-callsite-rel-freq`	Opt	60	Sets a minimum block frequency for a call site to be hot (no profile information)
`-inline-call-penalty`	Opt	25	Sets a call penalty per function invocation
`-inline-enable-cost-benefit-analysis`	Opt	false	Enables the cost-benefit analysis for the inliner
`-inline-cost-full`	Opt	false	Enables to compute the full inline cost when the cost exceeds the threshold
`-inline-caller-superset-nobuiltin`	Opt	true	Enables inlining when a caller has a superset of the callee's `nobuiltin` attribute
`-disable-gep-const-evaluation`	Opt	false	Disables the evaluation of `GetElementPtr` with constant operands

Asm2Vec

- -Oz and –Os does not receive inlining but it does receives optimization
- We find that Asm2Vec significantly misclassified function pairs including -Oz