C Source Code Analysis for Memory Safety

Sound Static Analysis for Security Workshop
NIST, June 26-27

Henny Sipma
Kestrel Technology

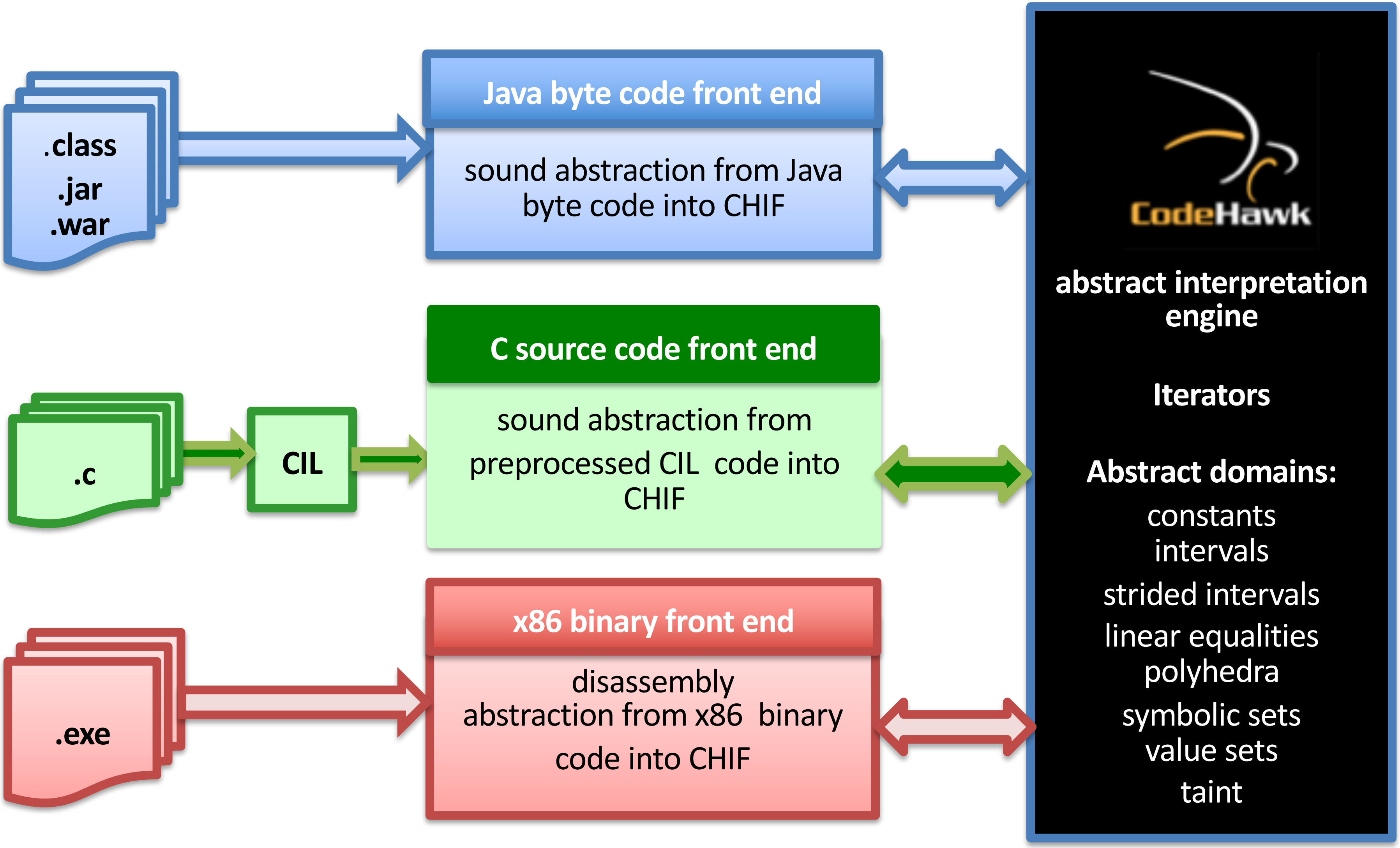
Kestrel Technology

Founded:	2000
Location:	Palo Alto, CA
Core activity:	Sound Static Analysis of Software
Languages supported:	C source, Java bytecode, x86 executables
Underlying technology:	Abstract interpretation (Cousot & Cousot, 1977)

Properties:

C:	Memory safety analysis
Java:	Information flow analysis, complexity analysis
x86:	Memory safety, information extraction, malware analysis, reverse engineering

Kestrel Technology CodeHawk Tool Suite



.class
.jar
.war

Java byte code front end

sound abstraction from Java byte code into CHIF

.c

CIL

C source code front end

sound abstraction from preprocessed CIL code into CHIF

.exe

x86 binary front end

disassembly
abstraction from x86 binary code into CHIF

CodeHawk

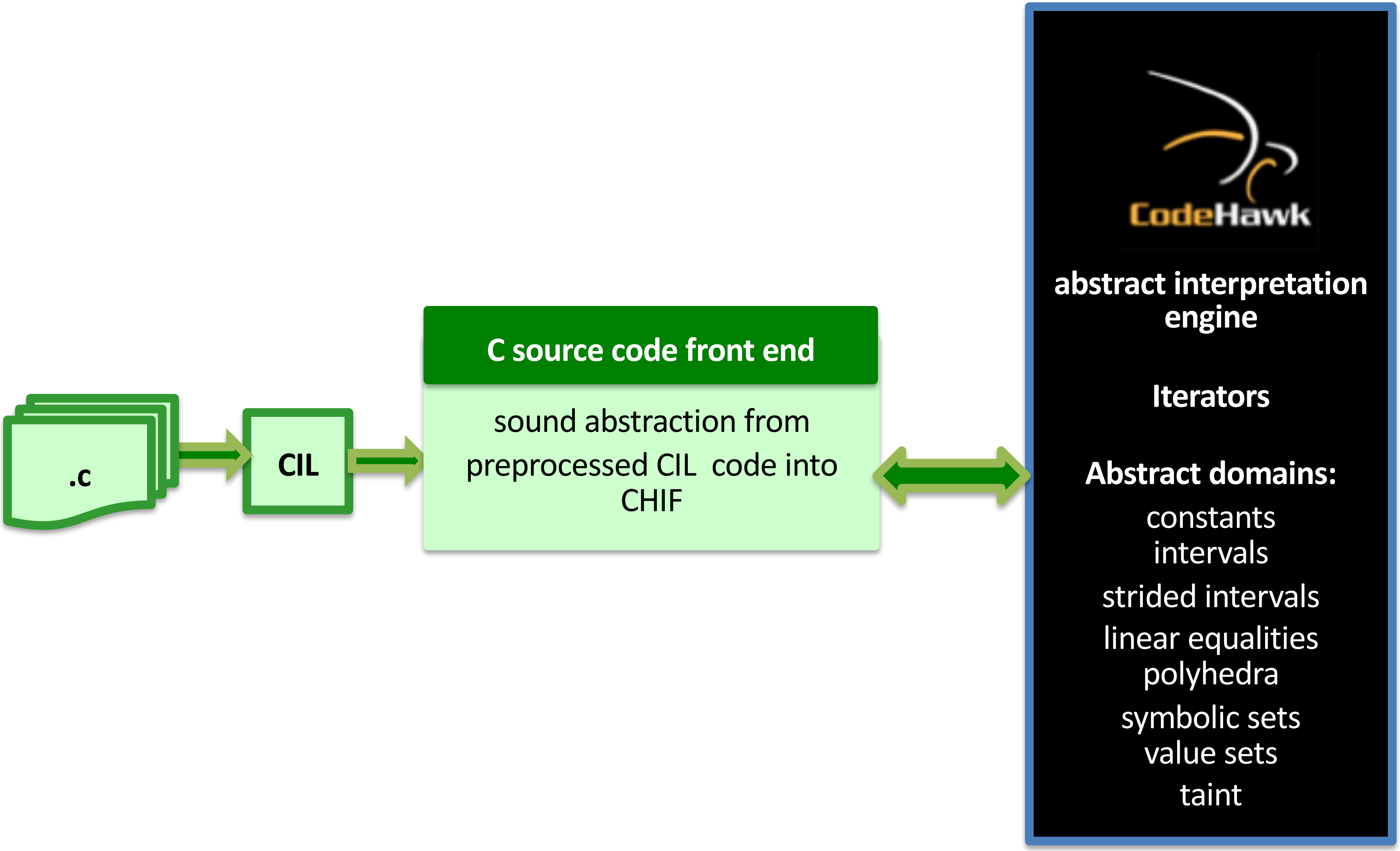
abstract interpretation engine

Iterators

Abstract domains:

constants
intervals
strided intervals
linear equalities
polyhedra
symbolic sets
value sets
taint

CodeHawk C Analyzer



.c

CIL

C source code front end

sound abstraction from preprocessed CIL code into CHIF

abstract interpretation engine

Iterators

Abstract domains:

constants
intervals
strided intervals
linear equalities
polyhedra
symbolic sets
value sets
taint

Sound Static Memory Safety Analysis for C

Goal: Mathematically prove absence of memory safety vulnerabilities (covering more than 50 CWEs) for real-world applications

Approach:

- **Specification**: C Standard – specification of undefined behavior
- Translate into preconditions on instructions and library functions
- Prove that all preconditions are valid

Advantages:

- If successful: full assurance of memory safety
- Exhaustive: no false negatives
- Evidence: results can be independently audited
- Metrics: progress and success

Sound Static Memory Safety Analysis for C
Challenges

Not automatic

May involve significant effort

Approach:

- ✓ **Specification**: C Standard – specification of undefined behavior
- ✓ Translate into preconditions on instructions and library functions
- ➢ Prove that all preconditions are valid

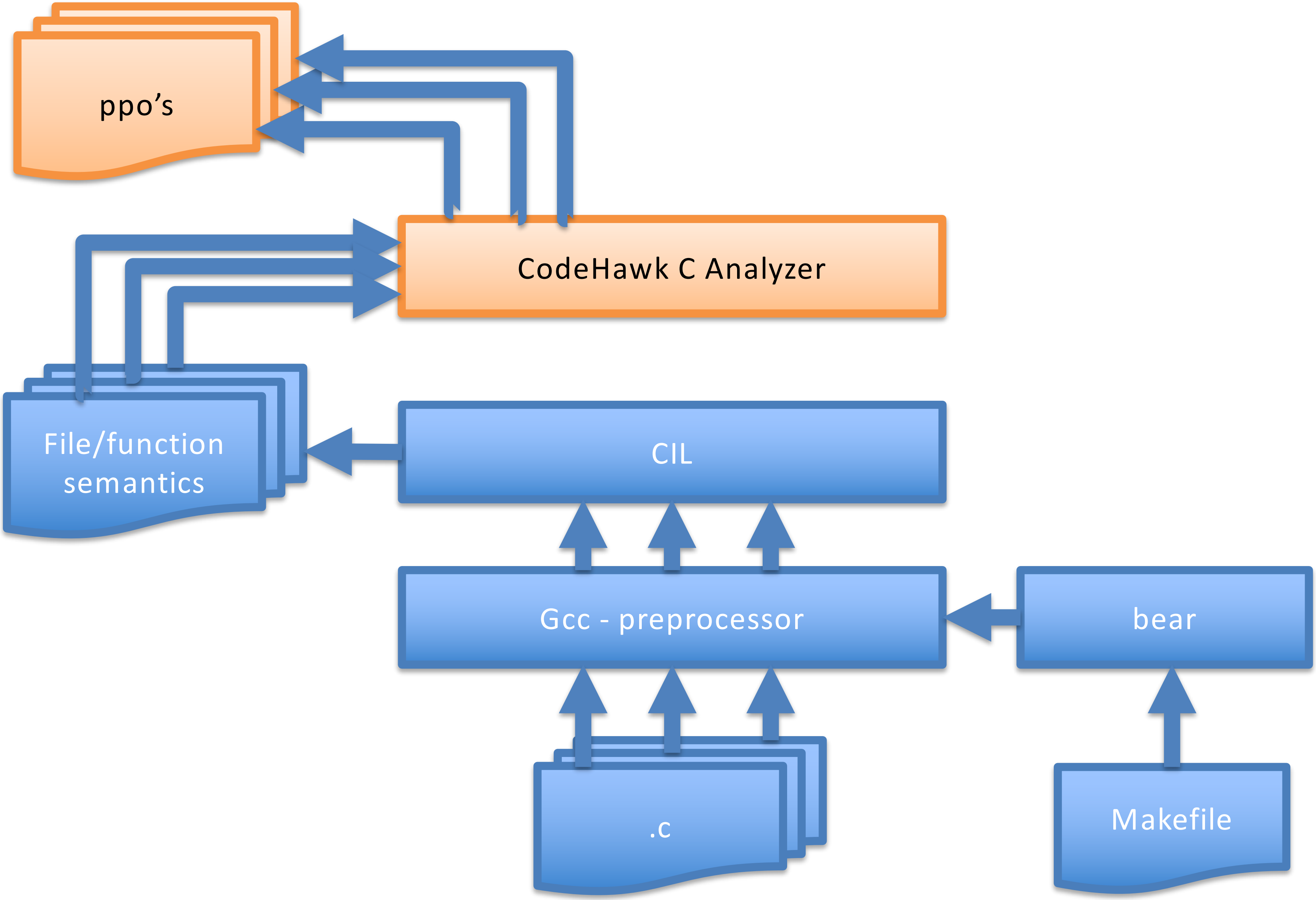
Open
Primary proof obligations
(ppo's)

Closed ppo's

Test Applications

application	LOC
Cairo-1.14.12	227,818
Cleanflight-CLFL-v2.3.2	118,758
Dnsmasq-2.76	29,922
Dovecot-2.0.beta6 (SATE 2010)	208,636
File	14,379
Git-2.17.0	205,636
Hping	11,336
Irssi-0.8.14 (SATE 2009)	61,972
Lighttpd-1.4.18 (SATE 2008)	49,747
Nagios-2.10 (SATE 2008)	47,652
Naim-0.11.8.3.1 (SATE 2008)	25,759
Nginx-1.14.0	103,388
Nginx-1.2.9	102,151
Openssl-1.0.1.f	275,060
Pvm3.4.6 (SATE 2009)	60,029
Wpa_supplicant-2.6	96,554
Total	**1,638,797**

Creating Primary Proof Obligations



ppo's

CodeHawk C Analyzer

File/function semantics

CIL

Gcc - preprocessor

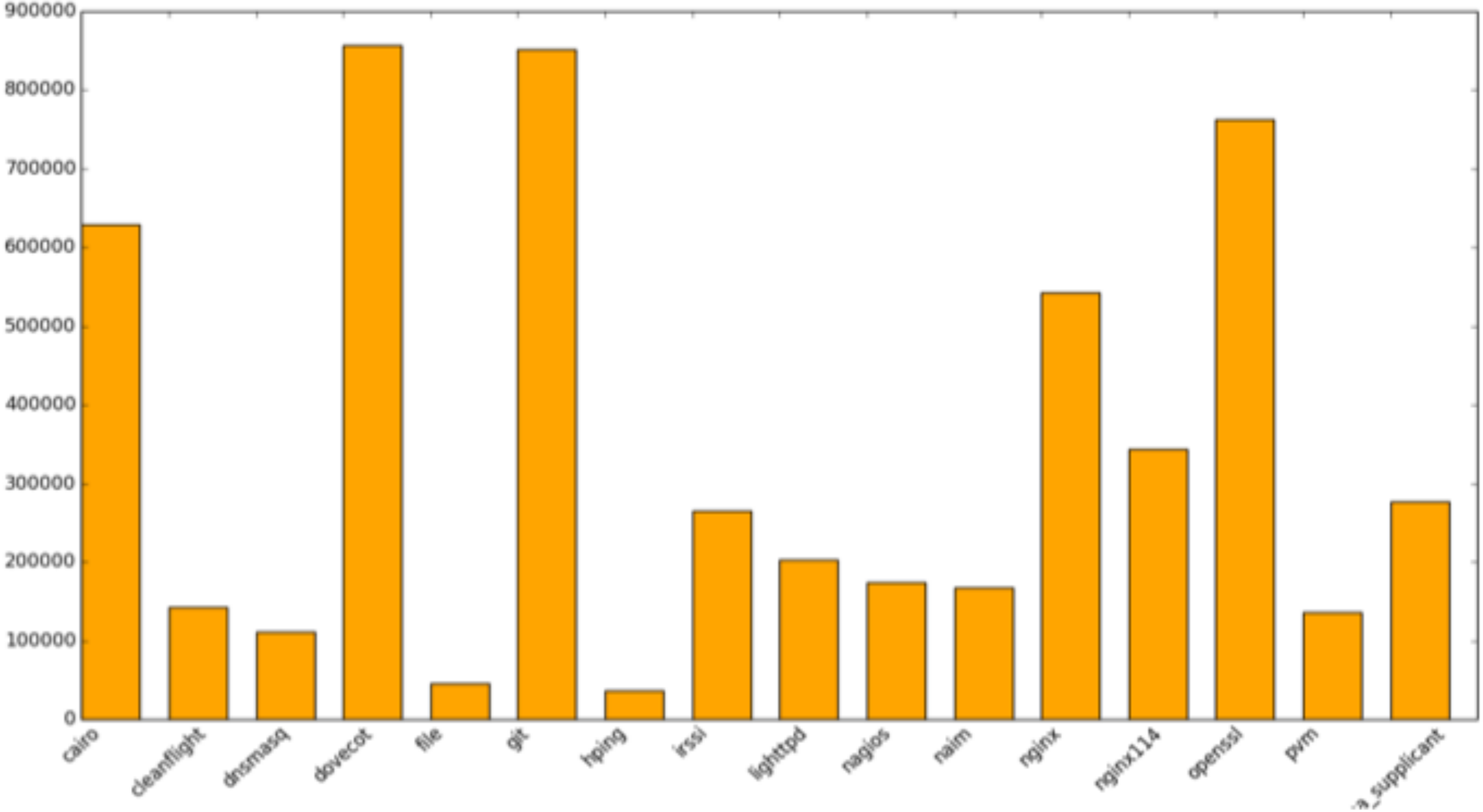
bear

.c

Makefile

Primary Proof Obligations: How Many?

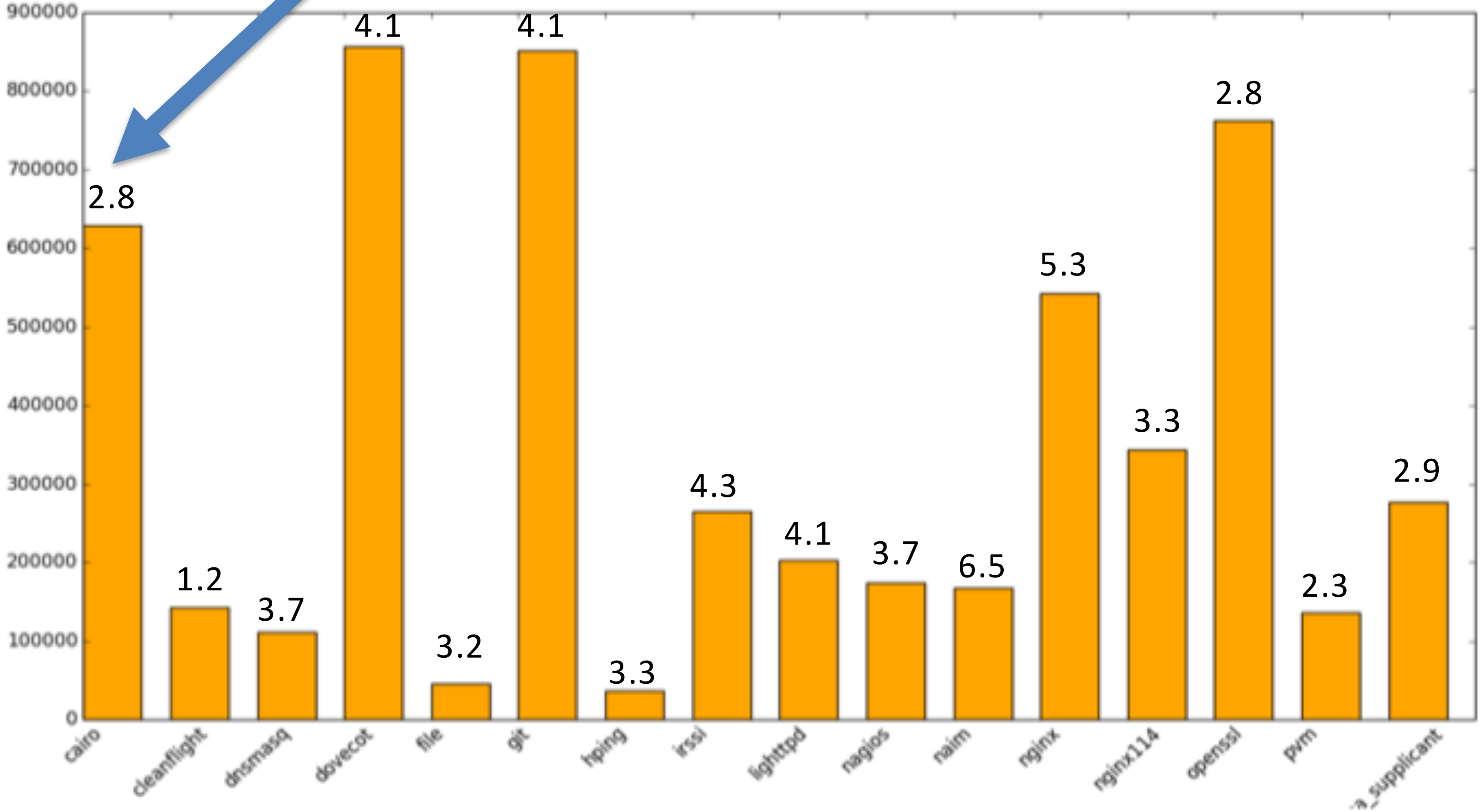
5,545,304



Primary Proof Obligations: How Many?

Ppo’s per lines of code

5,545,304



Primary Proof Obligations: What are they?

First-order atomic predicates:

- allocation-base(p)
- cast(x,t1,t2)
- common-base(p1,p2)
- common-base-type(p1,p2)
- format-string(p)
- global-memory(p)
- index-lower-bound(a)
- index-upper-bound(a,s)
- initialized(v)
- initialized-range(p,s)
- int-overflow(op,a,b,t)
- int-underflow(op,a,b,t)
- lower-bound(p)
- no-overlap(p1,p2)
- non-negative(a)
- not-null(p)
- not-zero(a)
- null(p)
- null-terminated(p)
- pointer-cast(p,t1,t2)
- ptr-lower-bound(op,p,a)
- ptr-upper-bound(op,p,a)
- ptr-upper-bound-deref(op,p,a)
- signed-to-unsigned-cast(a,t1,t2)
- unsigned-to-signed-cast(a,t1,t2)
- upper-bound(p)
- valid-memory(p)
- value-constraint(x)
- width-overflow(a)

Primary Proof Obligations: Analysis
Simple Things First

A. Check validity based on individual statement and declarations

```
int a[10];
…
a[3] = 0;
```

- index-lower-bound(3)
- index-upper-bound(3,10)

```
strcpy(dst,"string")
```

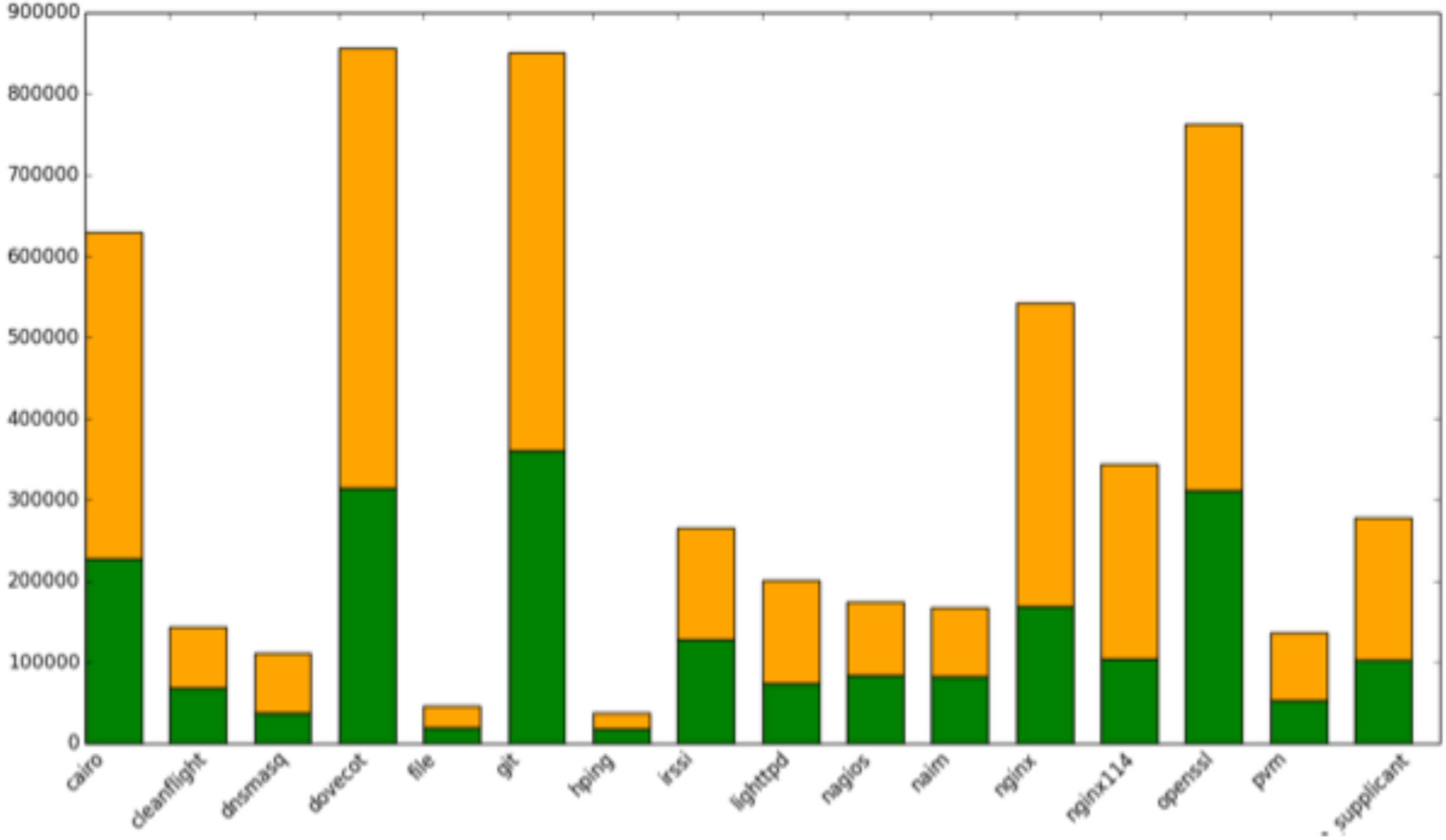
- null-terminated(“string”)
- not-null(“string”)
- lower-bound(“string”)
- upper-bound(“string”)
- valid-memory(“string”)

Primary Proof Obligations

3,389,371

2,155,365

Discharge ppo's at the statement level

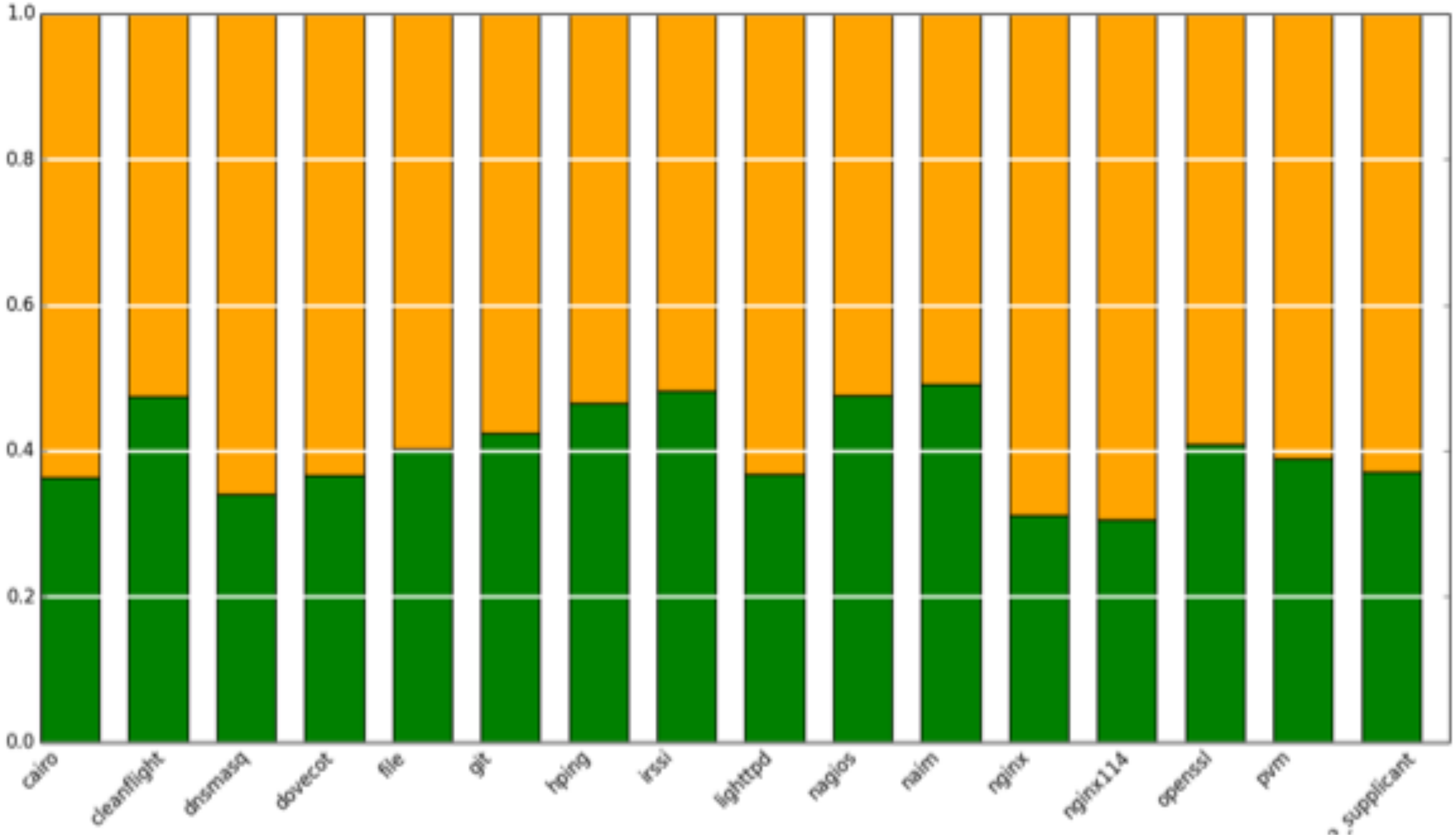


Primary Proof Obligations

Discharge ppo's at the statement level (as a percent of total)

3,389,371

2,155,365



Primary Proof Obligations: Analysis
Generating Invariants

B. Check validity based on invariants generated

```
int a[10];
....
for (int i=0; i < 10; i++) {
  a[i] = 0;
}
```

index-lower-bound(i)
index-upper-bound(i)

i = [0 .. 9]

```
 1. int x;
....
10. x = ....
....
20. x = x + 1;
```

...
initialized (x)
...

x:initialized@10

proof obligations

invariant

Analysis: Generating Local Invariants (Context-insensitive)

- Abstract Interpretation (Cousot,Cousot, 1977)
- Domains:
    - Intervals (Cousot, Cousot)
    - Linear Equalities (Karr, 1976)
    - Value Sets (Reps, 2004)
    - Symbolic Sets
    - Parametric Ranges
- Flow-sensitive, Path-insensitive

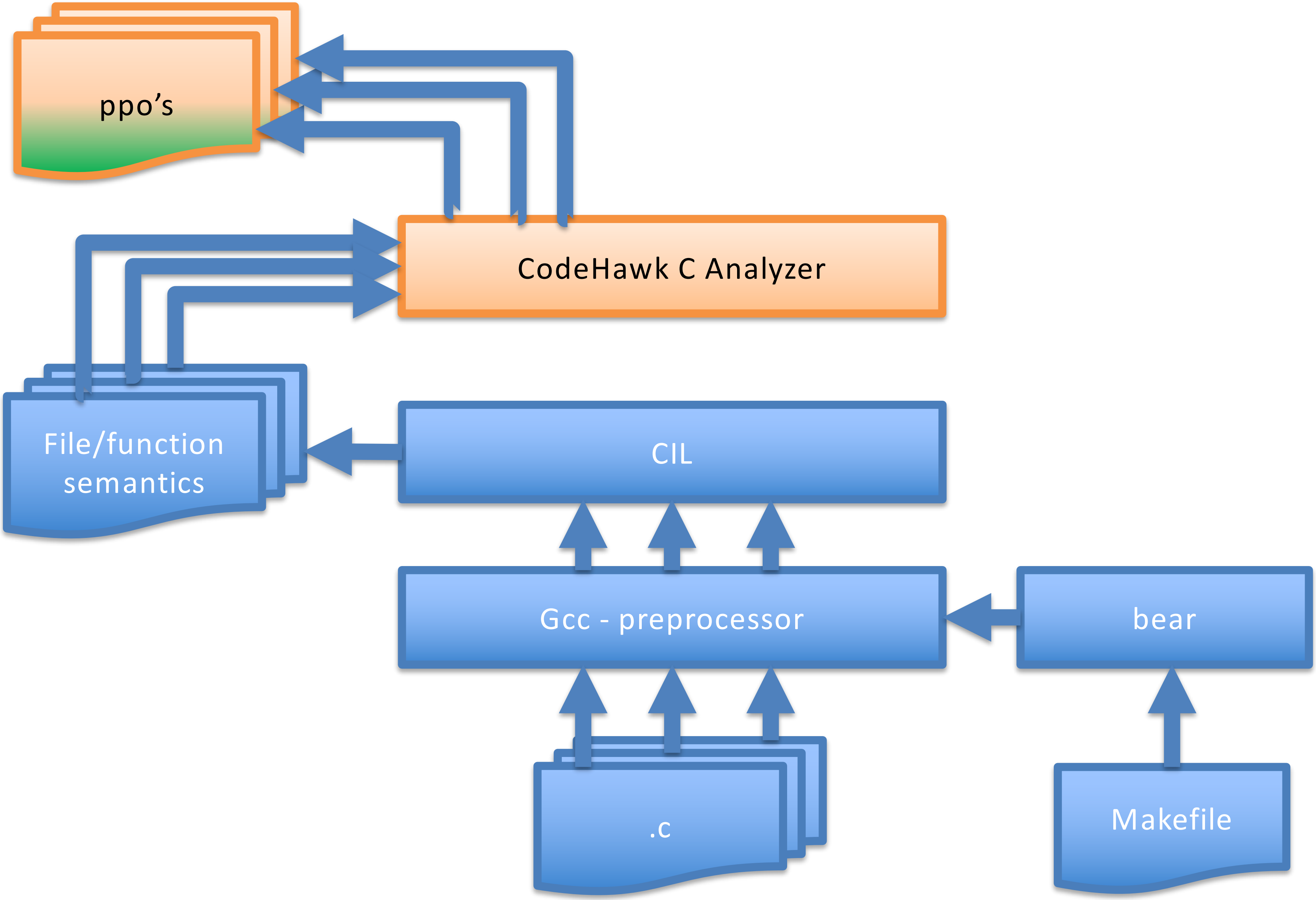
CodeHawk

abstract interpretation engine

Iterators

Abstract domains:
constants
intervals
strided intervals
linear equalities
polyhedra
symbolic sets
value sets
taint

Analysis: Generating Local Invariants



ppo’s

CodeHawk C Analyzer

File/function semantics

CIL

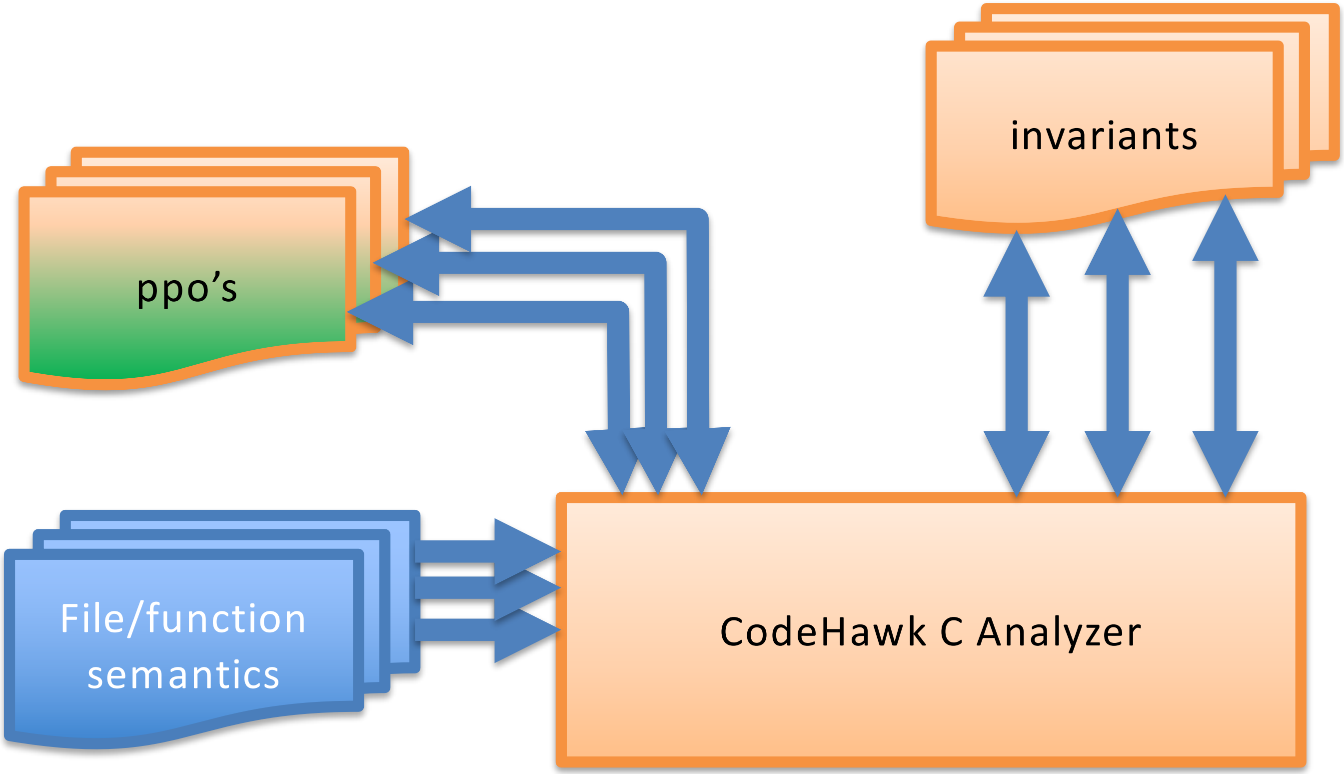
Gcc - preprocessor

bear

.c

Makefile

Analysis: Generating Local Invariants

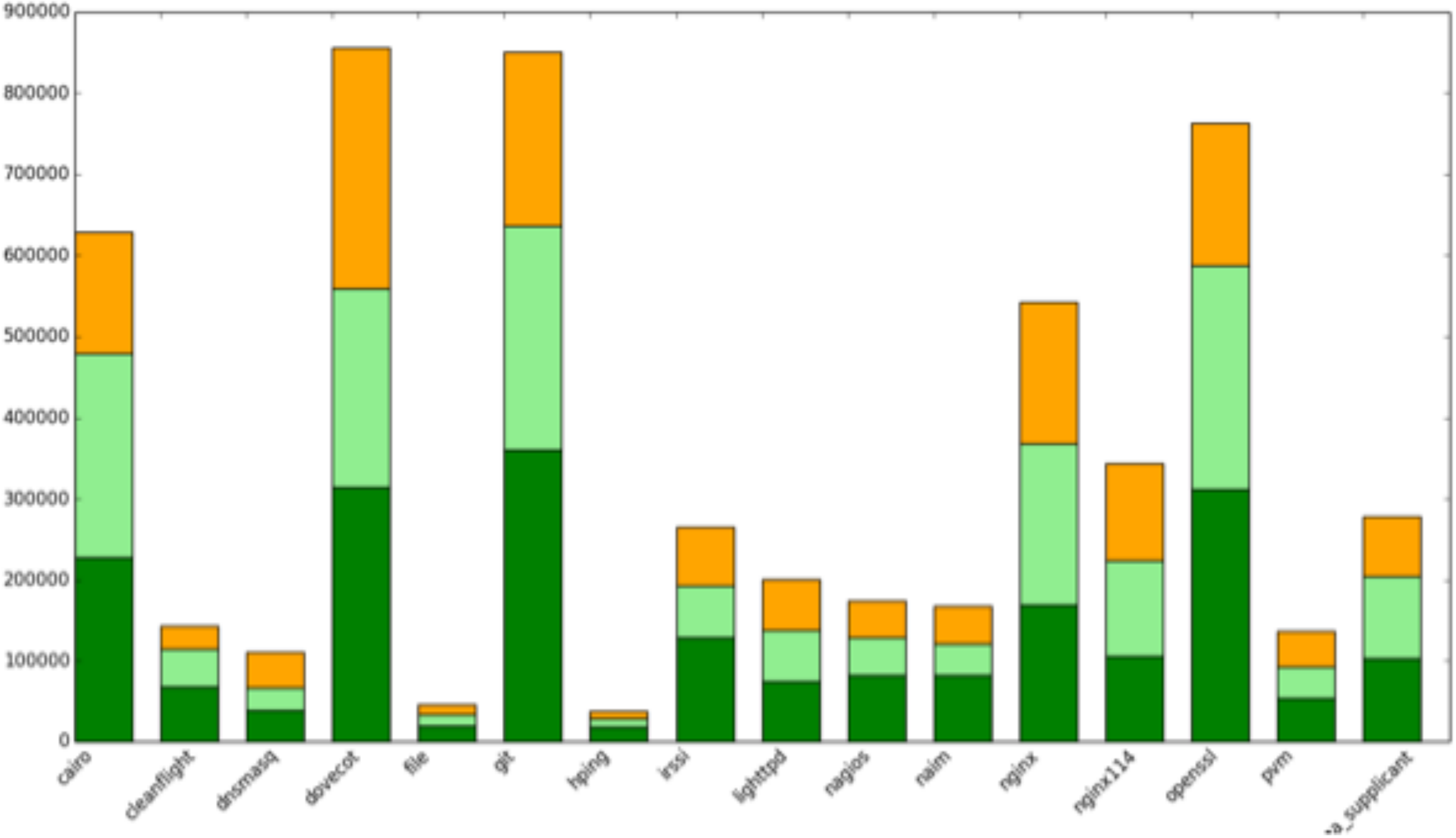


Primary Proof Obligations

Discharge ppo’s using local function invariants

1,568,639

3,976,097

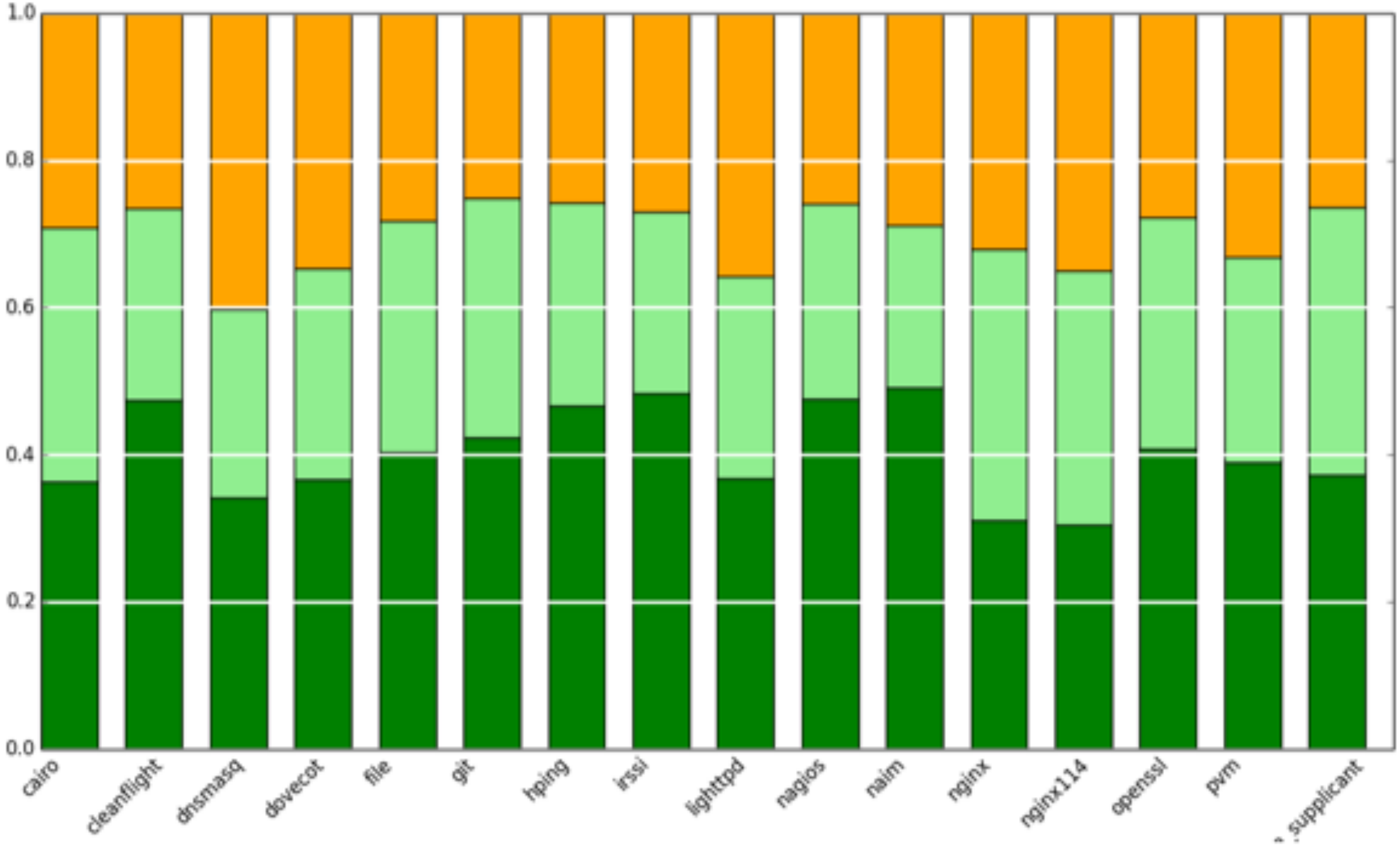


Primary Proof Obligations

Discharge ppo's using local function invariants

1,568,639

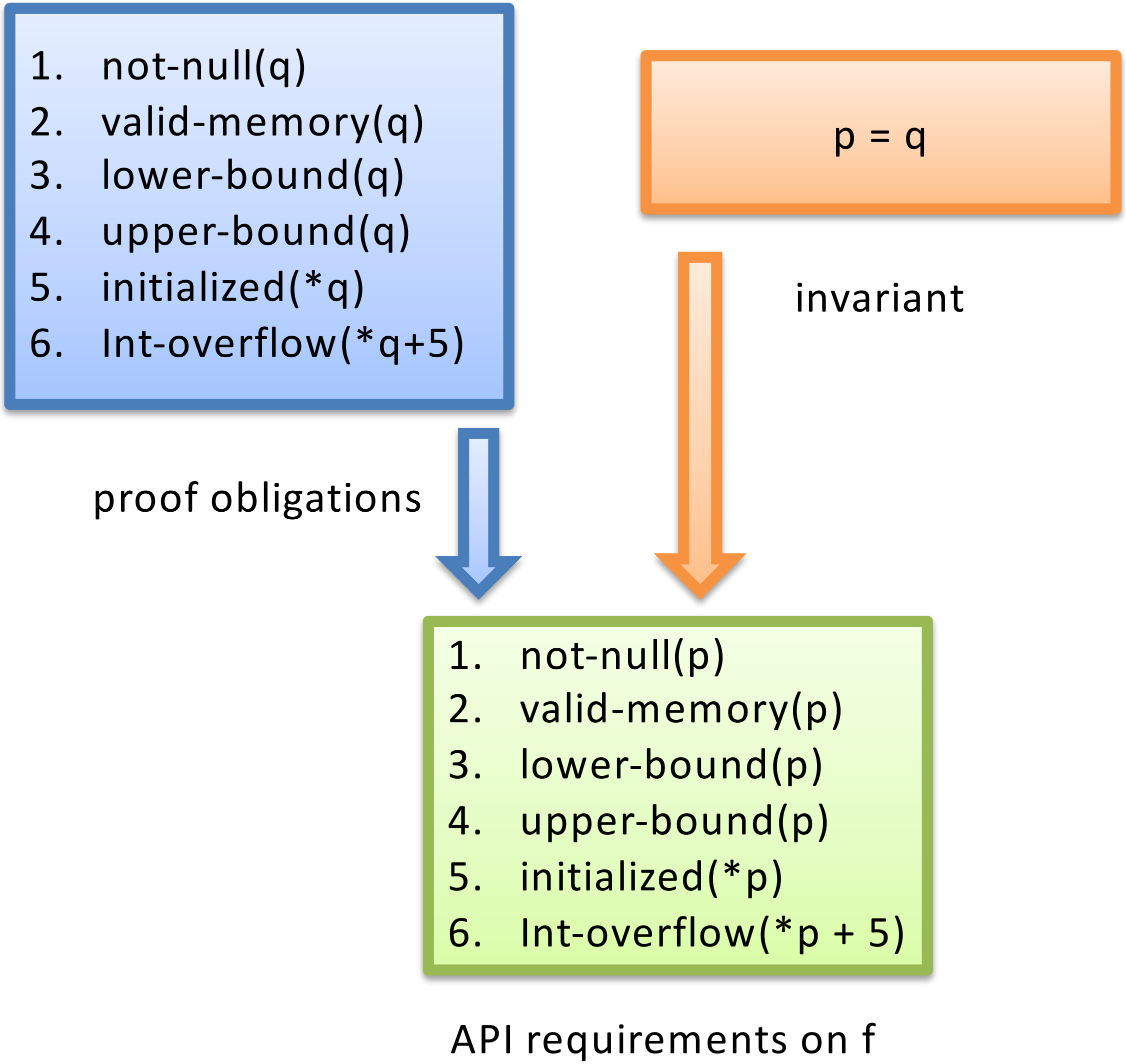
3,976,097



Analysis: Delegating Proof Obligations (Context sensitivity)

C. Lift responsibility to api

```
int f (int *p) {
  int *q;
  int x;
  ...
  q = p;
  x = *q + 5;
```



1. not-null(q)
2. valid-memory(q)
3. lower-bound(q)
4. upper-bound(q)
5. initialized(*q)
6. Int-overflow(*q+5)

p = q

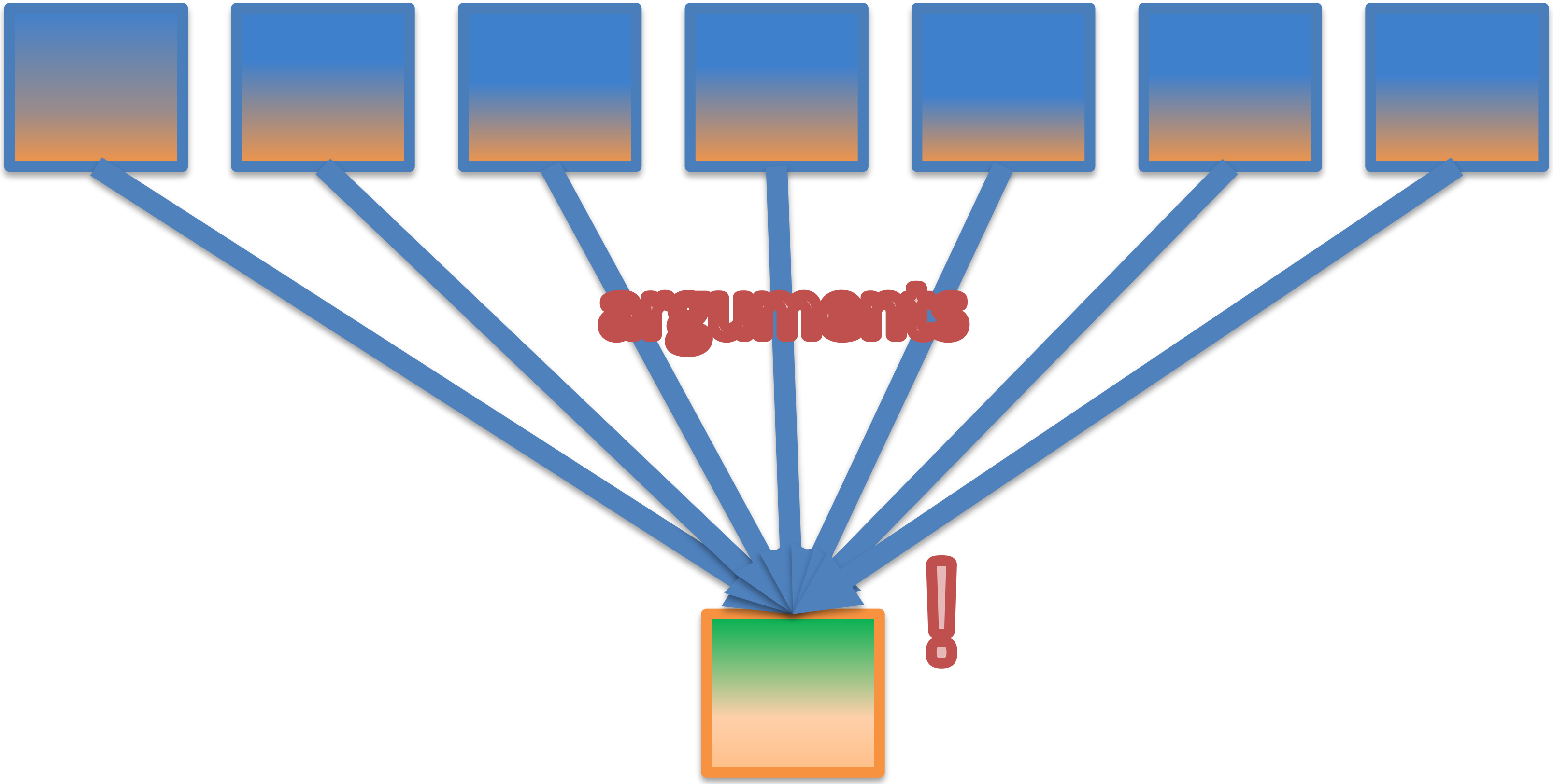
invariant

proof obligations

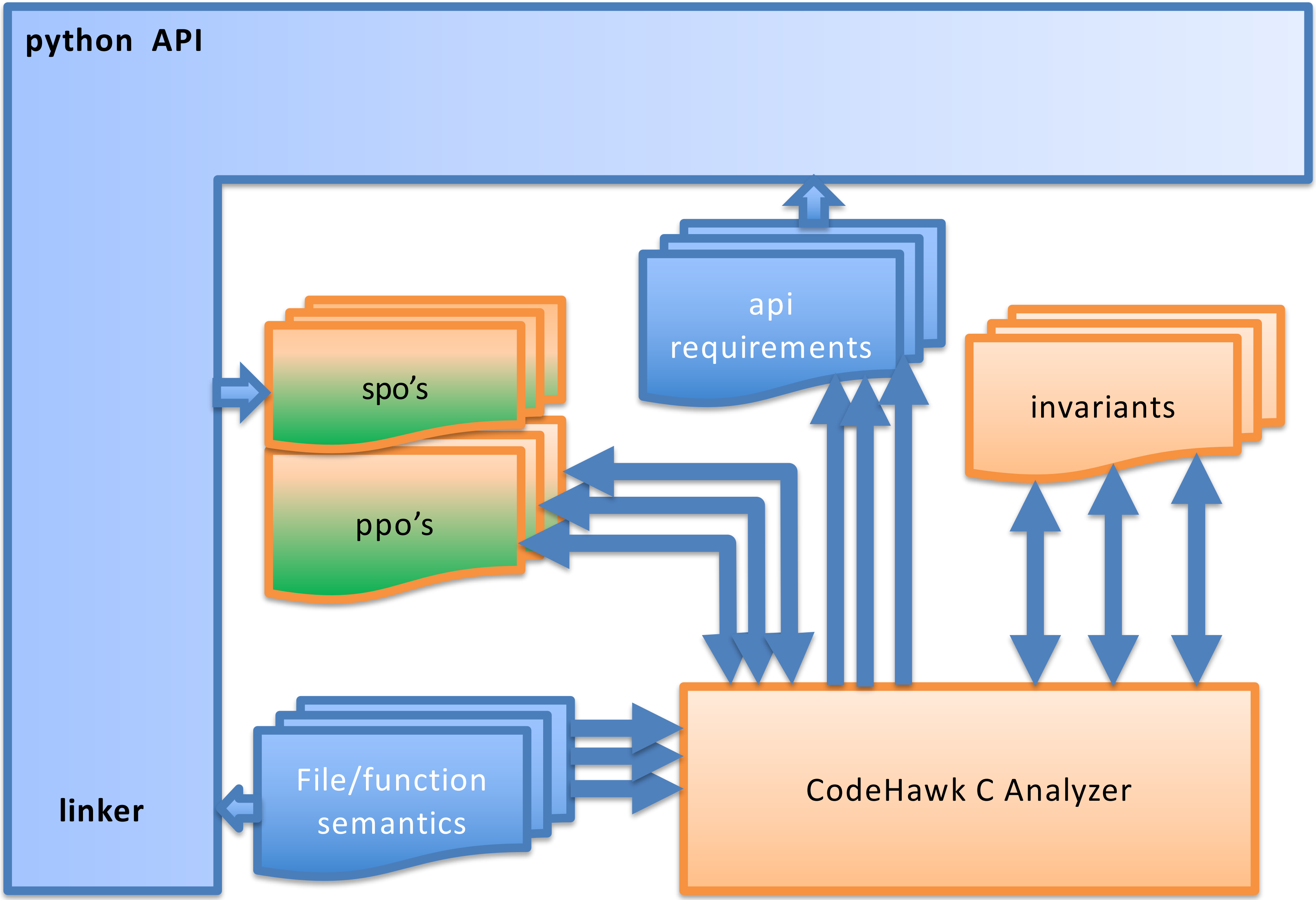
1. not-null(p)
2. valid-memory(p)
3. lower-bound(p)
4. upper-bound(p)
5. initialized(*p)
6. Int-overflow(*p + 5)

API requirements on f

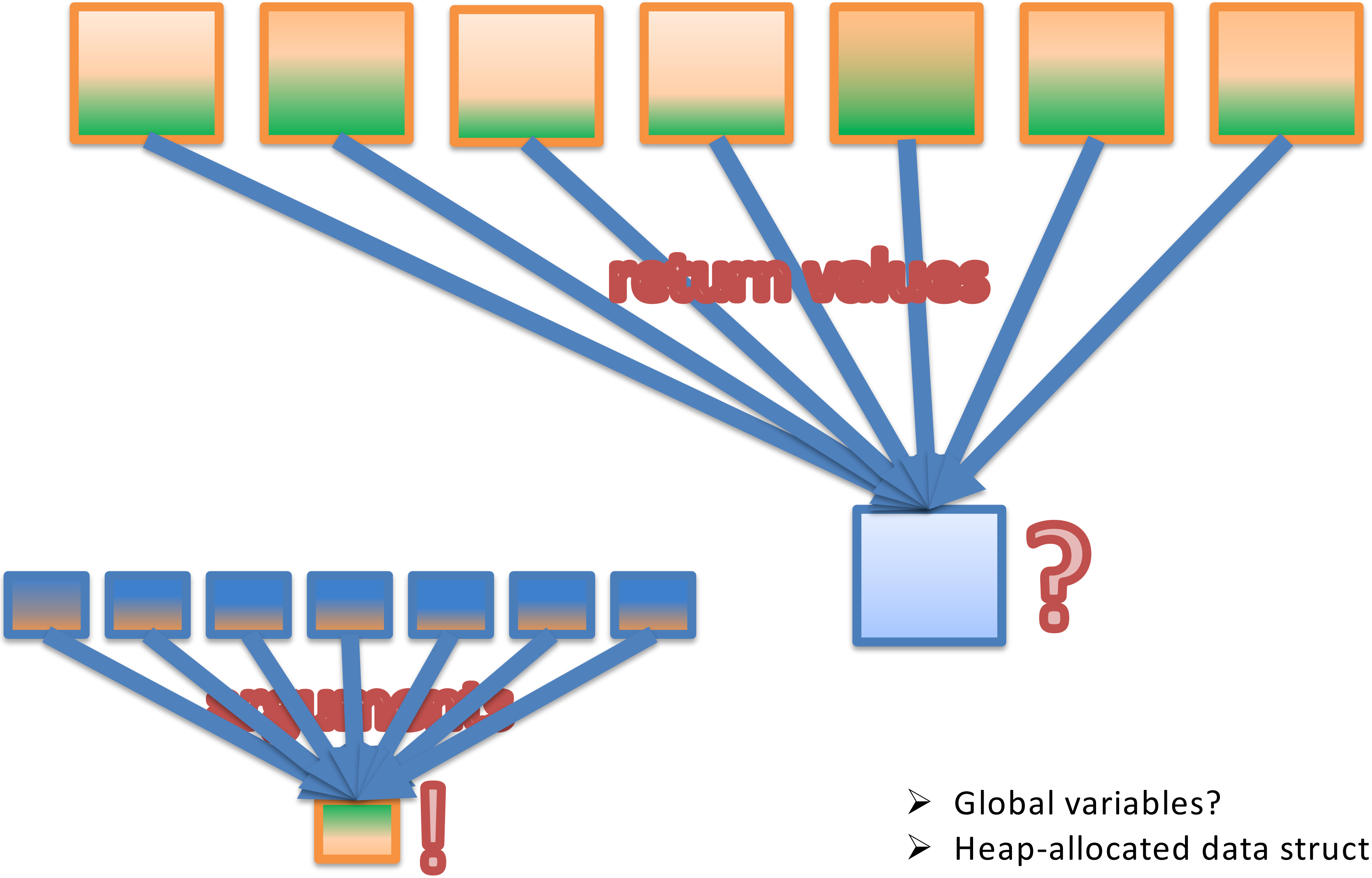
Analysis: Delegating Proof Obligations Impose Preconditions on Callers



Analysis: Delegating Proof Obligations
Create Supporting Proof Obligations

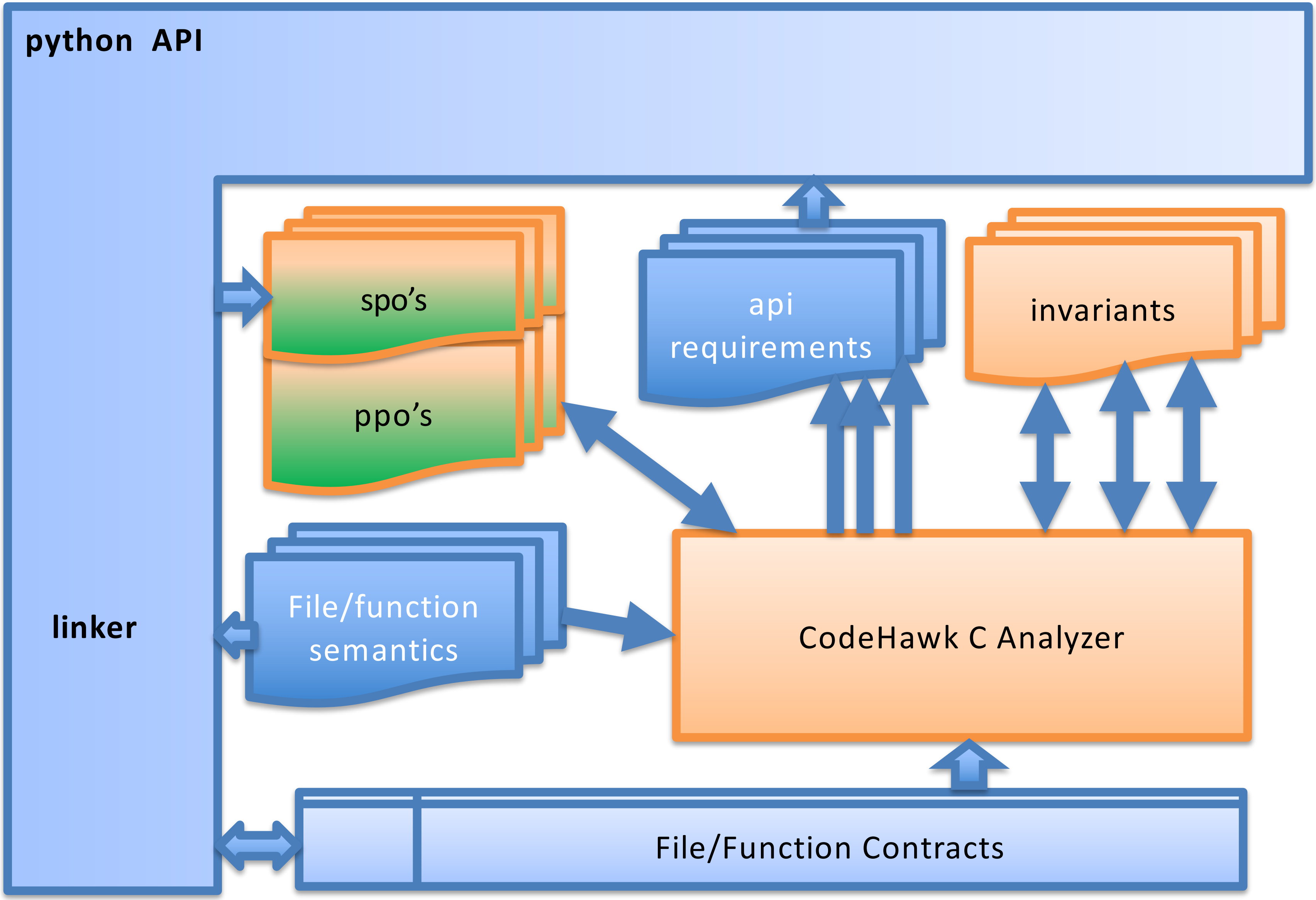


Analysis: Delegating Proof Obligations Impose Postconditions on Callers?

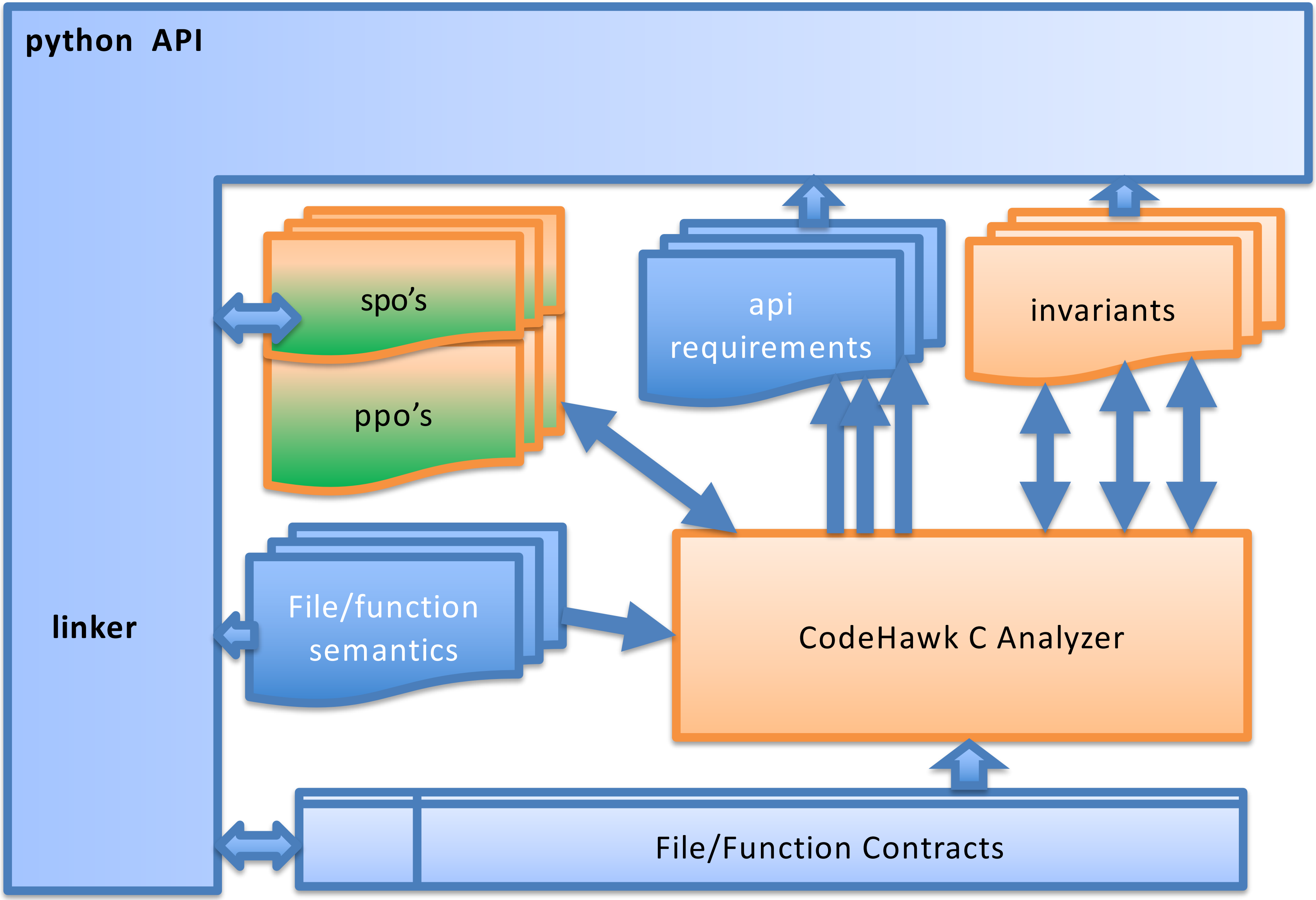


- Global variables?
- Heap-allocated data structures?

Analysis: Delegating Proof Obligations
File/Function Contracts



Analysis: Delegating Proof Obligations
File/Function Contracts

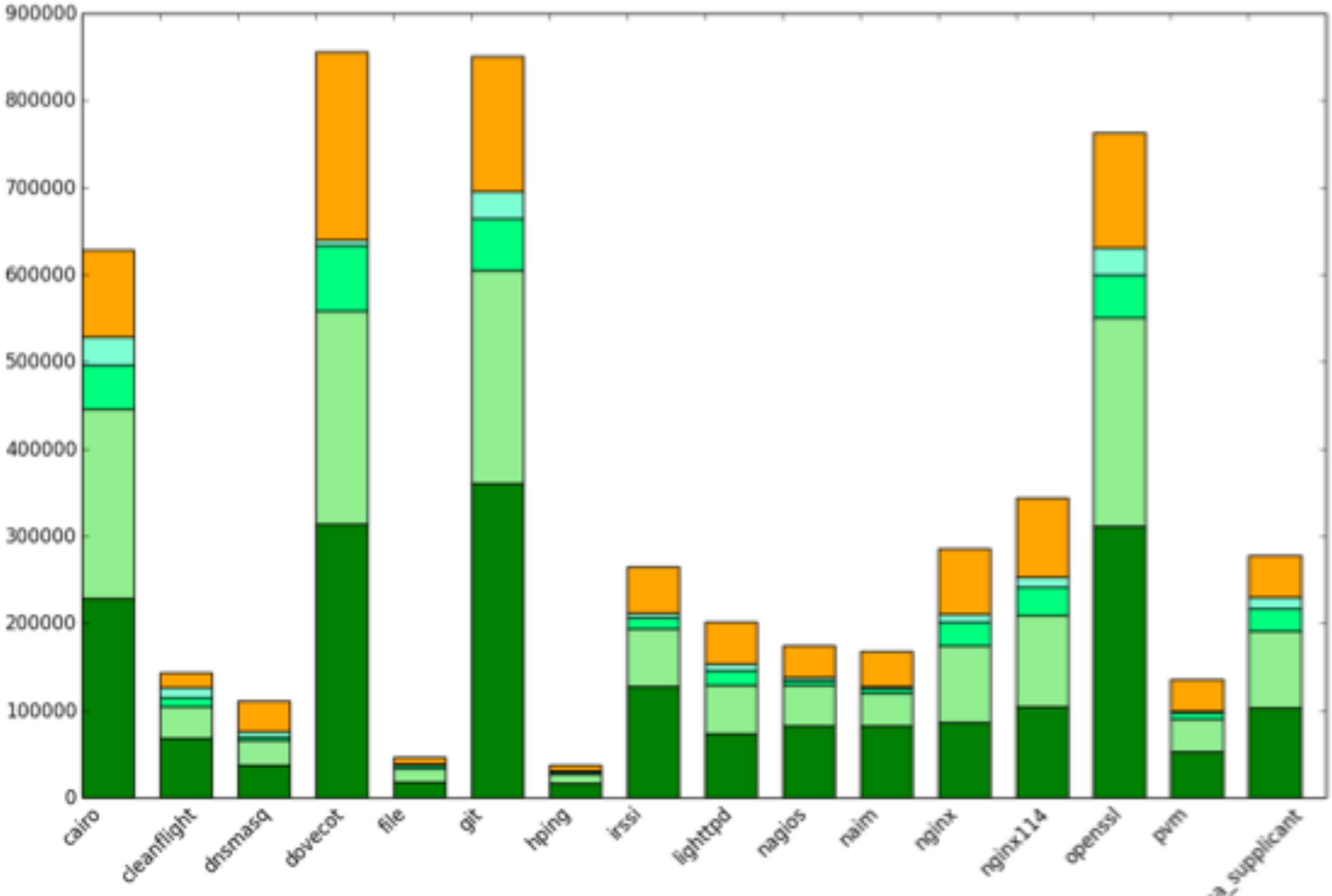


Primary Proof Obligations

Discharge ppo's using context sensitivity

1,097,471

4,191,788

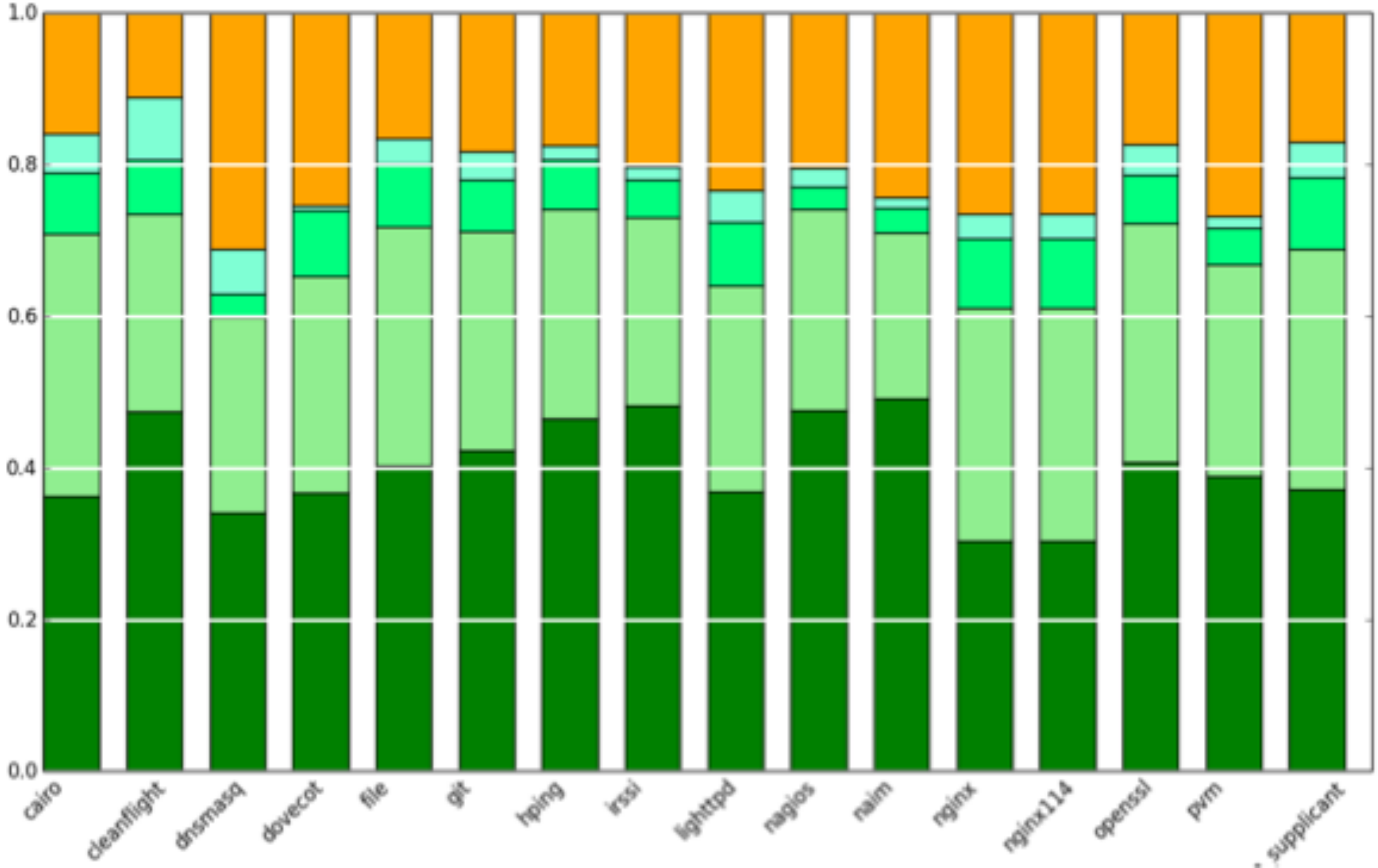


Primary Proof Obligations

Discharge ppo’s using context sensitivity

1,097,471

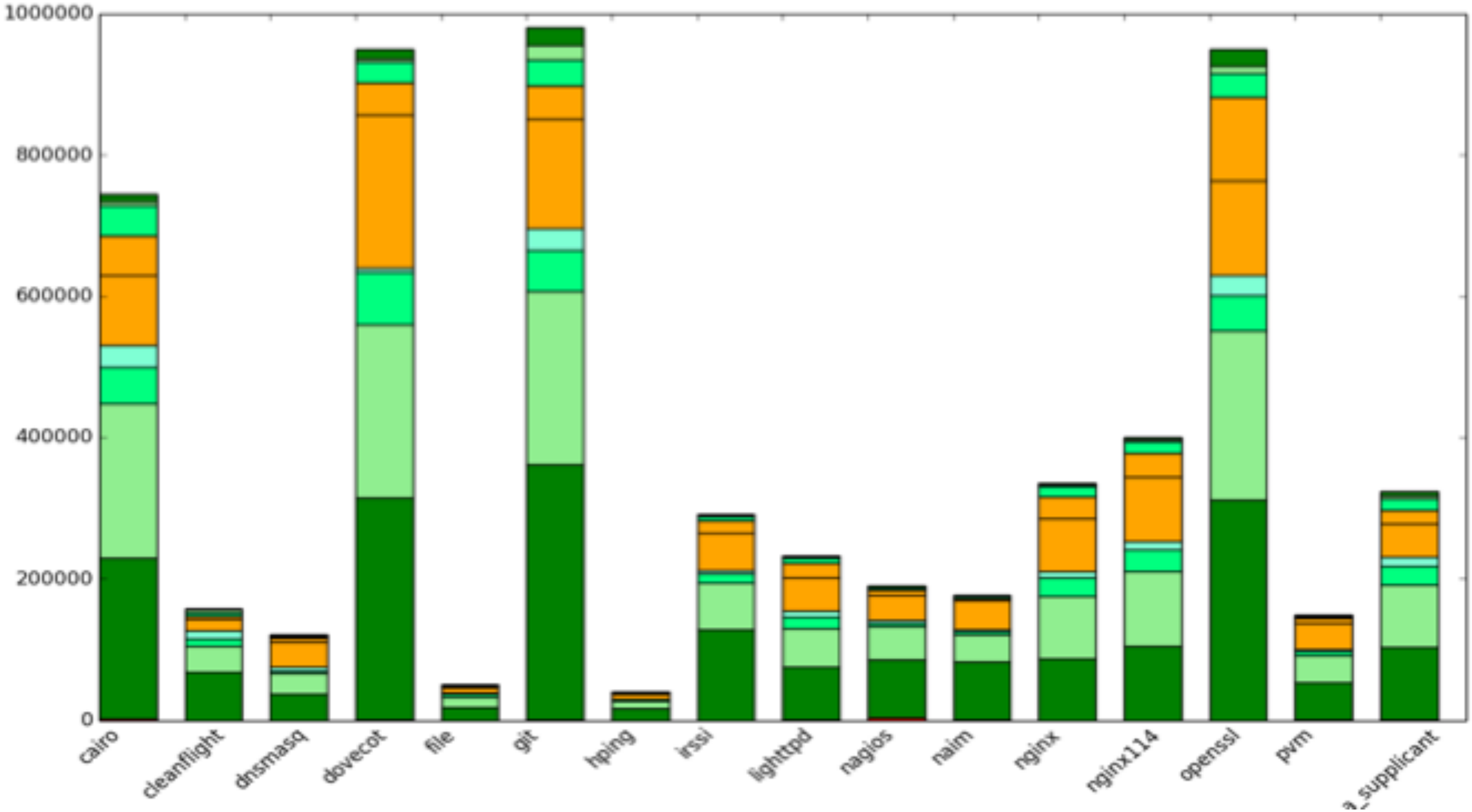
4,191,788



Primary + Supporting Proof Obligations

1,097,471 + 415,211 = 1,512,682

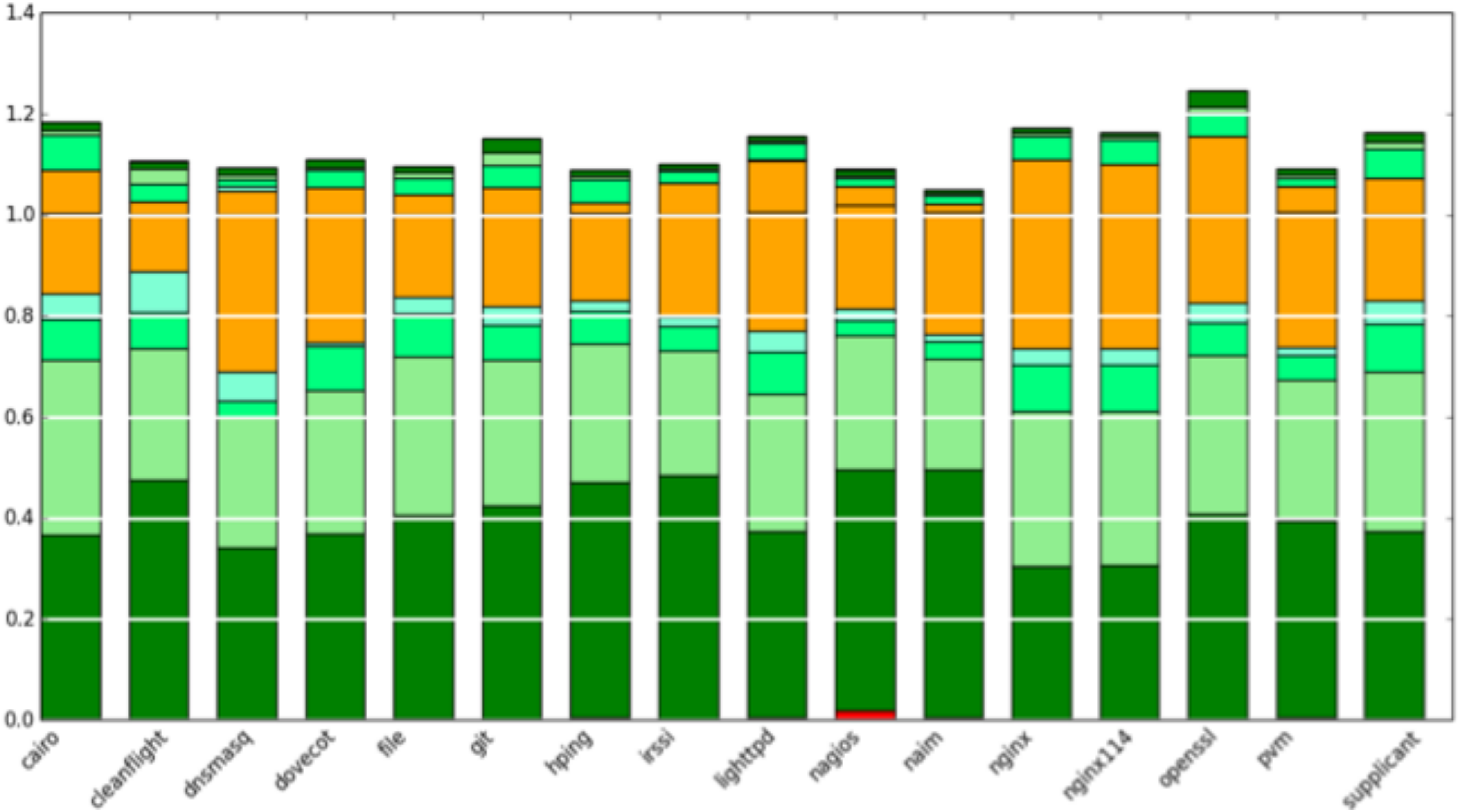
4,191,788 + 378,013 = 4,569,801



Primary + Supporting Proof Obligations

1,097,471 + 415,211 = 1,512,682

4,191,788 + 378,013 = 4,569,801



Bugs, False Positives?

Our perspective: Anything that cannot be proven safe needs work:

- ➢ Additional user input (in the form of contract conditions), and/or
- ➢ Additional analysis capabilities, and/or
- ➢ Modifications to the program

A proof obligation is marked ‘violated’ (and closed) if

- the reason it cannot be proven safe is known, and
- no additional information can make it safe

Violations can indicate

- A bug, (primary or supporting proof obligation), or
- A contract condition that is too strong (supporting proof obligation), or
- A potential violation outside the analysis realm

Violations

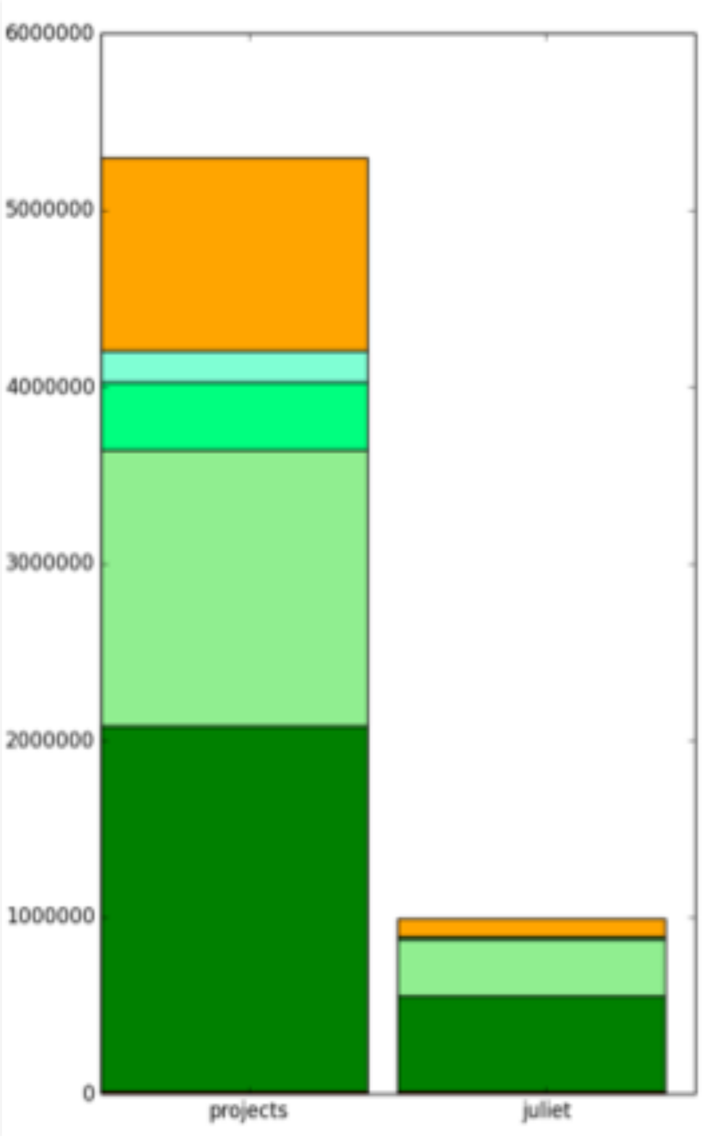
Mostly not very likely or not very interesting

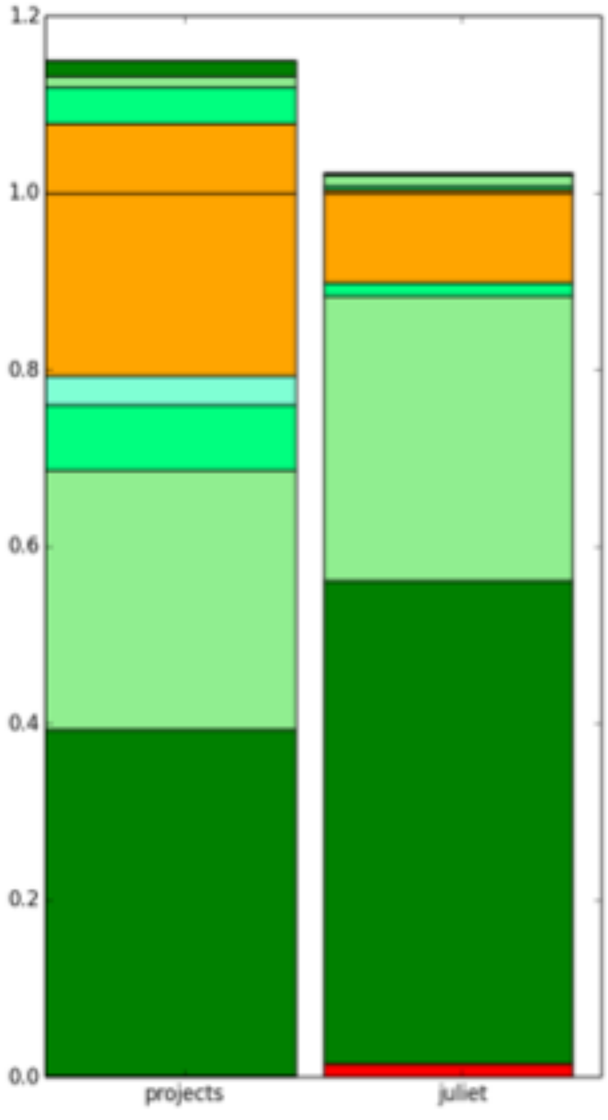
- A proof obligation is violated for all behaviors (universal)
- An existential condition is identified that violates a proof obligation
  - Use of return value from malloc, calloc, realloc without null check
  - Use of return value from fopen, getenv, etc., without null check
  - Cast -1 to unsigned integer
  - Unchecked user input values
  - Volatile values, random values
- An existential condition outside the realm of reasoning is identified that may violate a proof obligation
  - unchecked return value from strchr, strrchr, strtol, strtoll, etc.
  - ……

But, …

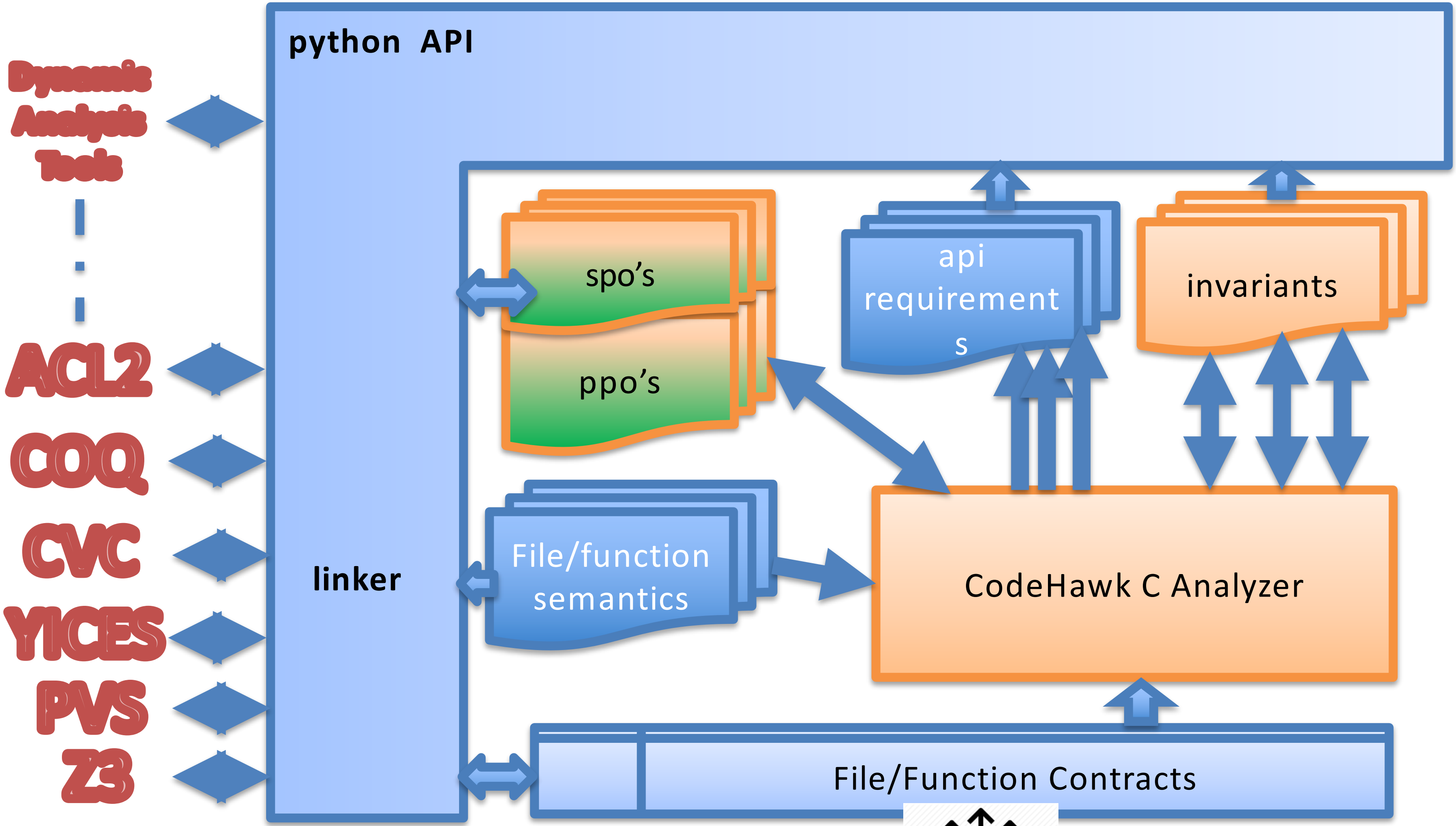
3 potentially serious memory vulnerabilities found in one of the test applications

Comparison between applications and Juliet Test Suite
~ 150 test cases





Collaborative Analysis of Open Source Software Tools and Communities



python API

Dynamic Analysis Tools

ACL2

COQ

CVC

YICES

PVS

Z3

linker

spo's

ppo's

api requirements

invariants

File/function semantics

CodeHawk C Analyzer

File/Function Contracts

SWAMP

SOFTWARE ASSURANCE MARKETPLACE

Conclusions

Our goal was:

Analysis

or

Open
Primary proof obligations
(ppo’s)

Closed ppo’s

Conclusions

Our goal was:

Analysis

or

Open
Primary proof obligations
(ppo’s)

Closed ppo’s

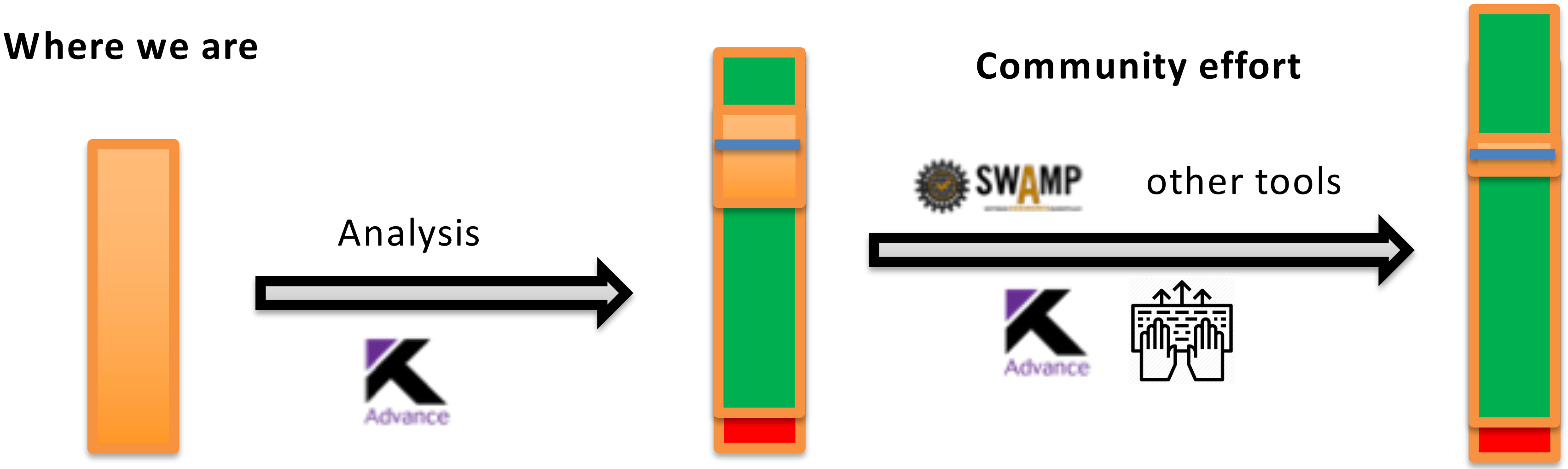
Where we are

Analysis

Supporting proof obligations

Primary proof obligations

Conclusions



Where we are

Analysis

Community effort

other tools

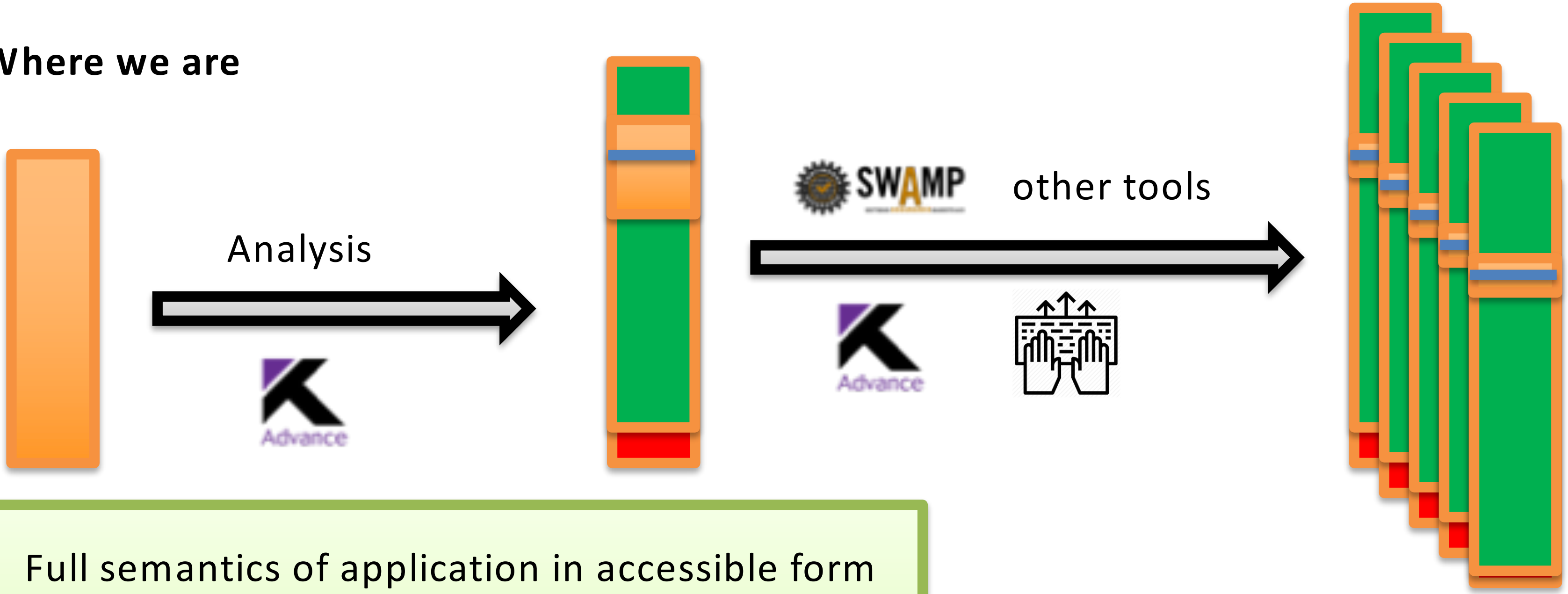
- Full semantics of application in accessible form
- Exhaustive set of proof obligations + evidence
- Function api conditions
- Invariants generated
- Programmable api in python

enables

- **Specialized analyses (academic tools)**
- **Incremental analysis**
- **Every result is subject to verification**
- **Modular analysis (function/file level)**
- **Clear measure of success**

Conclusions

Where we are



- Full semantics of application in accessible form
- Exhaustive set of proof obligations +evidence
- Function api conditions
- Invariants generated
- Programmable api in python

Most analysis results can be reused across versions;
Assumptions can be rechecked

enables

- **Specialized analyses (academic tools)**
- **Incremental analysis**
- **Every result is subject to verification**
- **Modular analysis (function/file level)**
- **Clear measure of success**

Conclusions: What's next?

- Extend with other properties, specified by state machines
- Extend expressiveness of contract specifications
- Continuous improvement of the analyzer, increase automation, C++
- Make C Analyzer available on the SWAMP

……….. and eventually (wishful thinking)

For every (many) important open-source C applications:

Create an open-source community-owned exhaustive set of proof obligations with (partial) analysis results, full set of assumptions (represented as api requirements and contract conditions) that evolves with new versions created

………… and (more wishful thinking)

Make sound static analysis an integral part of the open-source software development process

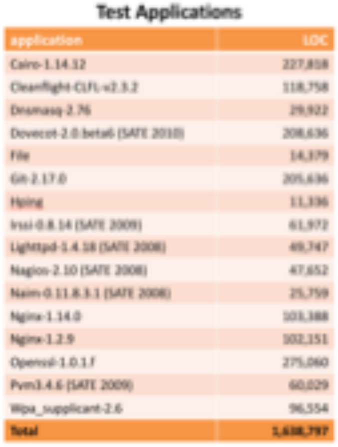
Conclusions: What's next?

Currently available on private GitHub repository

If you want to contribute contact us:

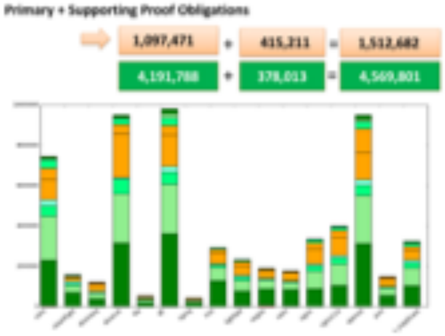
sipma@kestreltechnology.com

THANK YOU !



Test Applications

application	LOC
Cairo-1.14.12	227,818
Cleanflight-CLFL-v2.3.2	118,758
Dnsmasq-2.76	29,922
Dovecot-2.0.beta6 (SATE 2010)	208,636
File	14,379
Git-2.17.0	205,636
Hping	11,336
Irssi-0.8.14 (SATE 2009)	61,972
Lighttpd-1.4.18 (SATE 2008)	49,747
Nagios-2.10 (SATE 2008)	47,652
Naim-0.11.8.3.1 (SATE 2008)	25,759
Nginx-1.14.0	103,388
Nginx-1.2.9	102,151
Openssl-1.0.1.f	275,060
Pvm3.4.6 (SATE 2009)	60,029
Wpa_supplicant-2.6	96,554
Total	**1,638,797**



Primary + Supporting Proof Obligations

Property Specification

Property: Absence of memory vulnerabilities

C Standard lists 37 memory-related conditions that lead to undefined behavior

Property: Absence of memory-related undefined behavior

Primary Proof Obligations

- on language constructs
- on standard library functions

Proof by Structural Induction

Prove, by structural induction on the program, that every state of every computation is well-defined

Initially: the starting state of every computation is well-defined

Inductive step: For every operation in the program:

Assume the inductive hypothesis:
the starting state of the operation is well-defined

Prove: the resulting state after the operation is well-defined, according the C semantics

Conclude:

- every state of every computation is well-defined
- **absence of memory access violations**

Proof by Structural Induction

Prove, by structural induction on the program, that every state of every computation is well-defined

Initially: the starting state of every computation is well-defined

Inductive step: For every operation in the program:

Assume the inductive hypothesis:
the starting state of the operation is well-defined

Prove: the resulting state after the operation is well-defined, according the C semantics

Problem: Inductive hypothesis is not strong enough to prove the inductive step

Proof by Structural Induction

Solution: Use **abstract interpretation** to generate invariants to strengthen the inductive hypothesis

Inductive step: For every operation in the program:

Assume:

- the inductive hypothesis (state is well-defined), and
- invariants generated for the starting state of the operation

Prove:

- the resulting state after the operation is well-defined, according to the C semantics

Proof by Structural Induction

Solution: Use **abstract interpretation** to generate invariants to strengthen the inductive hypothesis

Domains:

- Intervals (Cousot & Halbwachs)
- Linear Equalities (Karr)
- Symbolic Sets
- Value sets (Balakrishnan, Reps)

Proof by Structural Induction

Prove, by structural induction on the program, that every state of every computation is well-defined

Approach is

- **Sound:** if all proof obligations can be proven valid, no memory access violations are possible
- **Complete:** if no memory access violations are possible then an inductive invariant exists to prove it

but (since undecidable)

not complete for demonstrating the existence of counter examples

CWE’s covered

118	Improper access of indexed resource (range error)
119	improper restriction of operations within the bound
120	Buffer copy without checking size of input (classic buffer overflow)
121	Stack-based buffer overflow
122	Heap-based buffer overflow
123	Write-what-where condition
124	Buffer underwrite
125	Out-of-bounds read
126	Buffer over-read
127	Buffer under-read
128	Wrap-around error
129	Improper validation of array index
130	Improper handling of length parameter inconsistency
131	Incorrect calculation of buffer size
135	Incorrect calculation of multi-byte string length
170	Improper null termination

CWE’s covered

190	Integer Overflow or wrap-around
191	Integer Underflow or wrap-around
193	Off-by-one error
195	Signed to unsigned conversion error
196	Unsigned to signed conversion error
242	Use of inherently dangerous function (as related to memory safety)
415	Double free
416	Use after free
456	Missing initialization of variable
466	Return of pointer value outside of expected range
467	Use of sizeof() on pointer type
469	Use of pointer subtraction to determine size
476	Null pointer dereference
588	Attempt to access child of non-structure pointer
590	Free of memory not on the heap
785	Use of path manipulation function without maximum-sized buffer

CWE’s covered

786	Access of memory location before start of buffer
787	Out-of-bounds write
788	Access of memory location after start of buffer
805	Buffer access with incorrect length value
822	Untrusted pointer dereference
823	Use of out-of-range pointer offset
824	Use of uninitialized pointer
825	Expired pointer dereference
839	Numeric range comparison check without maximum check
843	Access of reource using incompatible type (type confusion)
369	Divide by zero
134	Uncontrolled format string
197	Numeric truncation

Abstracting C into CHIF

Some constructs are representable precisely:

```
int x;
....
x = x + 1;
```

```
x := x + 1
```

Some constructs are not (yet) supported:

```
int x, y;
....
x =  | y;
```

```
abstract(x)
```

Abstracting C into CHIF

CHIF is a register language: no aliasing, no pointers

```
int x;
....
f(&x);
```

abstract(x)

```
int x;
int *p;
....
p = &x;
*p = *p + 1;
```

remove x from analysis

Abstracting C into CHIF

CHIF analysis is intra-procedural

use assume-guarantee reasoning for interprocedural relationships

```
int f(...) {
  ....
  if (...) {
   ...
   return 0;
  }
  ....
  return 1;
}
```

```
int x;
....
x = f(...);
```

x in [0..1]

Many more constructs in C that require specialized abstraction

Test Applications

application	LOC	PPO's
Cairo-1.14.12	227,818	628,808
Cleanflight-CLFL-v2.3.2	118,758	143,015
Dnsmasq-2.76	29,922	110,743
Dovecot-2.0.beta6 (SATE 2010)	208,636	856,210
File	14,379	46,209
Git-2.17.0	205,636	851,087
Hping	11,336	37,079
Irssi-0.8.14 (SATE 2009)	61,972	265,345
Lighttpd-1.4.18 (SATE 2008)	49,747	202,157
Nagios-2.10 (SATE 2008)	47,652	173,868
Naim-0.11.8.3.1 (SATE 2008)	25,759	167,533
Nginx-1.14.0	103,388	343,759
Nginx-1.2.9	102,151	542,697
Openssl-1.0.1.f	275,060	762,621
Pvm3.4.6 (SATE 2009)	60,029	136,320
Wpa_supplicant-2.6	96,554	277,853
Total	**1,638,797**	**5,545,304**