Automatically Reasoning About the Cache Usage of Network Stacks



Rishabh Iyer Katerina Argyraki George Candea

Problem Statement

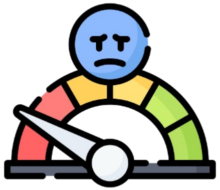
- Help developers answer
  - **Frequently-asked, what-if questions** about cache usage of code
  - Particularly for **unseen and untested workloads**

- Example questions
  - How does cache usage scale with the number of connections?
  - What is my code’s cache hit/miss profile?

Motivating Example

- Alice wants to build a fast, in-memory key-value store
  - Hash table + network stack (off-the-shelf)
  - Throughput bottlenecked by L3 cache misses

- Alice needs to answer questions such as
  - What workloads lead to consistent cache misses?
  - How much of the cache does each component use?



Existing Tools are Insufficient!

- Developers rely on profilers and HW counters today
- No predictive capability, insights limited to the **concrete** inputs used
- Developers must manually reverse engineer answers to key questions
  - Tedious and error-prone, particularly for third-party code

Recent patch showed how Linux's TCP stack had been incurring a bloated cache footprint, leading to slowdowns of up to 45%

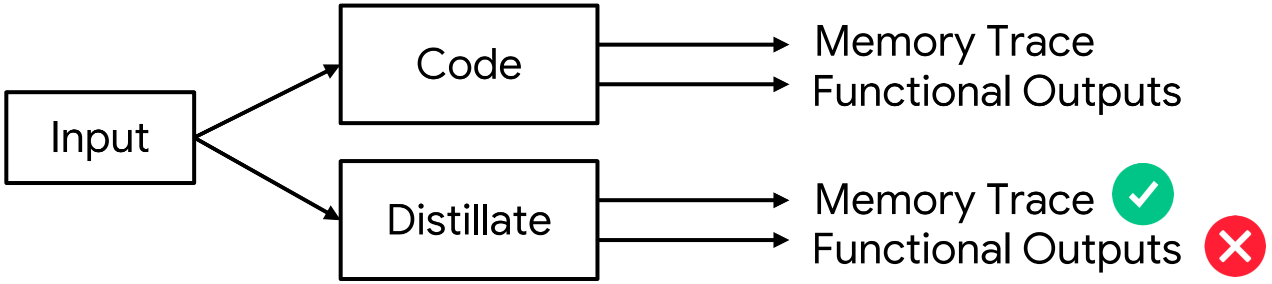
A Lack Of Abstraction For Cache Usage

- Alice needs visibility into how the code processes an **abstract/symbolic** workload
- Only way to obtain this information today is to read/profile the **implementation**

Can there exist an abstract/symbolic representation that helps developers efficiently reason about cache usage?

Memory Distillates

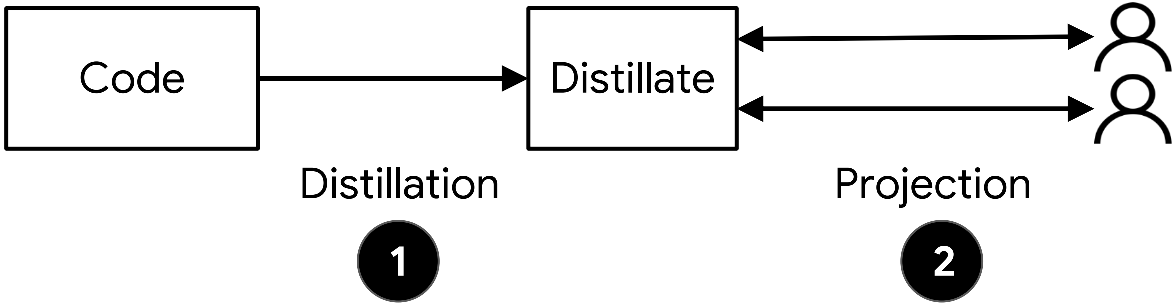
- Representation that retains all information relevant to how the code accesses memory
  - Discards everything else
- Given the same inputs as the code, the distillate
  - Produces an identical trace of memory accesses
  - But does not produce correct outputs



Cache Footprint AnalyzeR (CFAR)

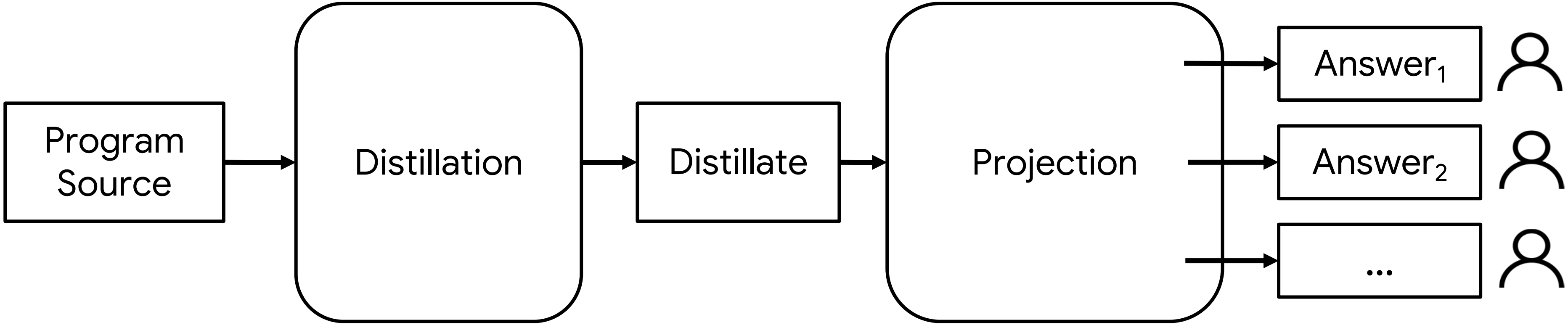
- Answers questions about cache usage using two-step workflow
  - Distillation: Extracts distillate using automated program analysis
  - Projection: Devs query distillate to answer specific questions

- Since distillate is precise, CFAR can answer diverse questions about cache usage

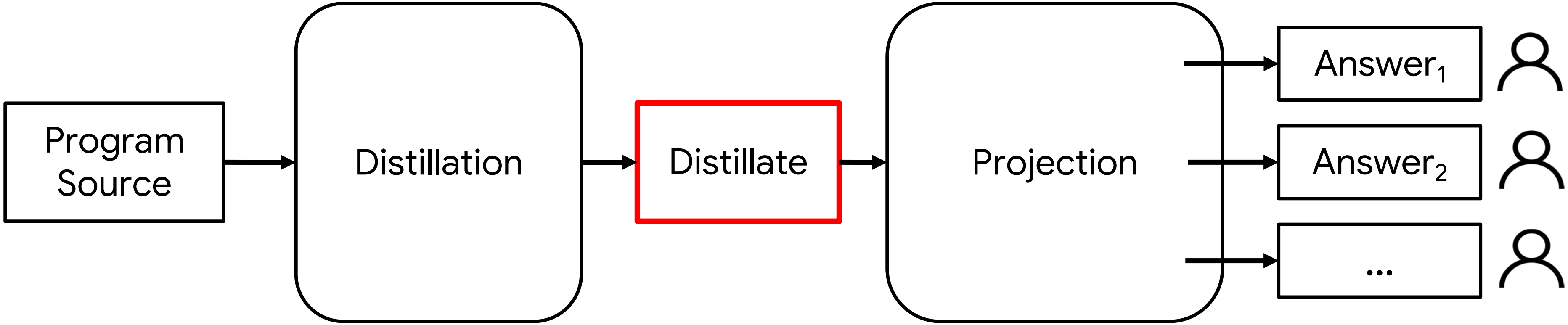


Memory distillates provide a simple yet precise abstraction for reasoning about cache usage

CFAR Overview



CFAR Overview



Example Syscall

- `sys_create()` from Hyperkernel

```
int sys_create(int fd, fn_t fn, uint64_t type,
               uint64_t value, uint64_t omode) {
    // State: pid, proc_tbl, file_tbl
    // Checking for invalid inputs
    if (type == FD_NONE) return -EINVAL;
    if (&proc_tbl[pid]->ofile[fd] != 0) return -EINVAL;
    struct file* file = &file_tbl[fn];
    if (file->refcnt != 0) return -EINVAL;

    // Update state
    file->type = type;
    file->value = value;
    file->omode = omode;
    file->refcnt = file->offset = 0;
    set_fd(pid, fd, fn);
    return 0;
}
```

Example Syscall

`sys_create()` from Hyperkernel

Kernel state: proc_table, filetable
Implemented as arrays

Input-dependent access pattern

```
int sys_create(int fd, fn_t fn, uint64_t type,
               uint64_t value, uint64_t omode) {
    // State: pid, proc_tbl, file_tbl
    // Checking for invalid inputs
    if (type == FD_NONE) return -EINVAL;
    if (&proc_tbl[pid]->ofile[fd] != 0) return -EINVAL;
    struct file* file = &file_tbl[fn];
    if (file->refcnt != 0) return -EINVAL;

    // Update state
    file->type = type;
    file->value = value;
    file->omode = omode;
    file->refcnt = file->offset = 0;
    set_fd(pid, fd, fn);
    return 0;
}
```

CFAR Distillate: Data Cache

```
def sys_create_dcache(fd, fn, type, value, omode):
    # State: pid, proc_tbl, file_tbl

    if type == FD_NONE: #6 accesses
      return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]

    if [proc_table+256*pid+64+8*fd]: #7 accesses
        return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd)
                ,..,(r,rsp-8)]
    .....
    # Succesful create. 17 accesses
    return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,
            (r,file_tbl+40*fn+8),(w,file_tbl+40*fn), (w,file_tbl+40*fn+16),
            ..,(w,proc_tbl+256*pid+64+8*fd),..,(r,rsp-8)]
```

CFAR Distillate: Data Cache

The data cache distillate of a program P is a program P^{data}_{dist}

P^{data}_{dist} takes the same inputs as P (I) and maintains the same state (S)

```
def sys_create_dcache(fd, fn, type, value, omode):
    # State: pid, proc_tbl, file_tbl

    if type == FD_NONE: #6 accesses
      return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]

    if [proc_table+256*pid+64+8*fd]: #7 accesses
        return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd)
                ,..,(r,rsp-8)]
    .....
    # Succesful create. 17 accesses
    return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,
            (r,file_tbl+40*fn+8),(w,file_tbl+40*fn), (w,file_tbl+40*fn+16),
            ..,(w,proc_tbl+256*pid+64+8*fd),..,(r,rsp-8)]
```

CFAR Distillate: Data Cache

P_{dist}^{data} returns an ordered sequence of data memory accesses Ω_{data}

Each memory access is a tuple `<type,addr>`

```
def sys_create_dcache(fd, fn, type, value, omode):
    # State: pid, proc_tbl, file_tbl

    if type == FD_NONE: #6 accesses
      return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]

    if [proc_table+256*pid+64+8*fd]: #7 accesses
        return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd)
                ,..,(r,rsp-8)]
    .....
    # Succesful create. 17 accesses
    return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,
            (r,file_tbl+40*fn+8),(w,file_tbl+40*fn), (w,file_tbl+40*fn+16),
            ..,(w,proc_tbl+256*pid+64+8*fd),..,(r,rsp-8)]
```

CFAR Distillate: Data Cache

type can be read (r), write (w), or read-modify-write (rmw)

addr is a symbolic function of I,S

```
def sys_create_dcache(fd, fn, type, value, omode):
    # State: pid, proc_tbl, file_tbl

    if type == FD_NONE: #6 accesses
      return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]

    if [proc_table+256*pid+64+8*fd]: #7 accesses
        return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd)
                ,..,(r,rsp-8)]
    .....
    # Succesful create. 17 accesses
    return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,
            (r,file_tbl+40*fn+8),(w,file_tbl+40*fn), (w,file_tbl+40*fn+16),
            ..,(w,proc_tbl+256*pid+64+8*fd),..,(r,rsp-8)]
```

CFAR Distillate: Data Cache

type can be read (r), write (w), or read-modify-write (rmw)

addr is a symbolic function of I,S

```
def sys_create_dcache(fd, fn, type, value, omode):
    # State: pid, proc_tbl, file_tbl

    if type == FD_NONE: #6 accesses
      return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]

    if [proc_table+256*pid+64+8*fd]: #7 accesses
       return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd)
               ,..,(r,rsp-8)]
    .....
    # Succesful create. 17 accesses
    return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,
           (r,file_tbl+40*fn+8),(w,file_tbl+40*fn), (w,file_tbl+40*fn+16),
           ..,(w,proc_tbl+256*pid+64+8*fd),..,(r,rsp-8)]
```

Symbolic representation enables distillate to replicate P's memory accesses irrespective of the concrete values of input/state and address space randomization

CFAR Distillate: Instruction Cache

The i-cache distillate is also a program P_{dist}^{instr} with the same arguments as P

P_{dist}^{instr} returns an ordered sequence of instr accesses Ω_{instr}

```
def sys_create_icache(fd, fn, ftype, value, omode):
    # State: pid, proc_tbl, file_tbl
    # sys_create abbreviated as s

    if ftype == FD_NONE: # 10 instructions
      return [(r,s),..,(r,s+168),..,(r,s+176)]

    # Error paths elided for presentation clarity
    ......

    # Succesful create. 45 instructions
    return [(r,s),(r,s+8),..,(r,s+160),(r,s+168),(r,s+176)]
```

CFAR Distillate: Instruction Cache

Instr addresses are offsets relative to the address of the first instruction of containing function in P

```
def sys_create_icache(fd, fn, ftype, value, omode):
    # State: pid, proc_tbl, file_tbl
    # sys_create abbreviated as s

    if ftype == FD_NONE: # 10 instructions
      return [(r,s),..,(r,s+168),..,(r,s+176)]

    # Error paths elided for presentation clarity
    ......

    # Succesful create. 45 instructions
    return [(r,s),(r,s+8),..,(r,s+160),(r,s+168),(r,s+176)]
```

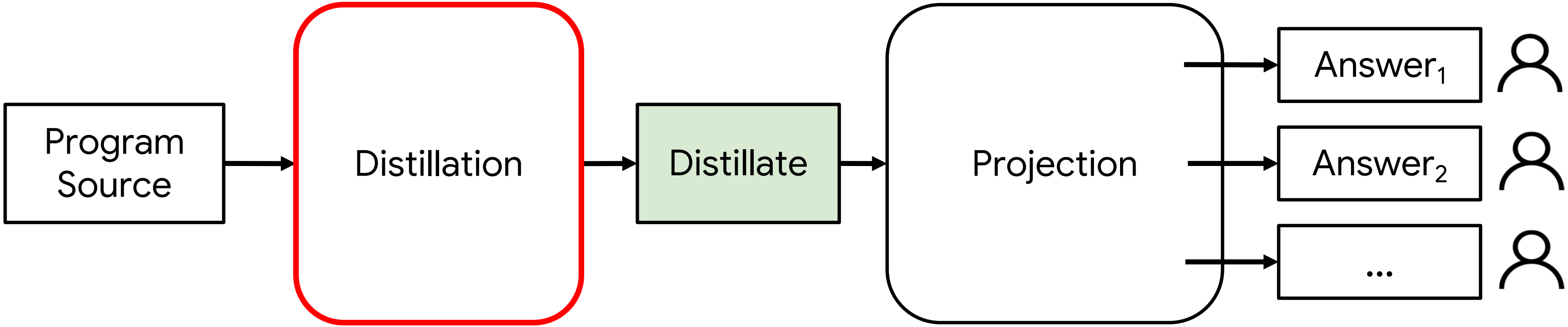
CFAR's i-cache distillate will produce the precise sequence of instructions executed by P irrespective of where the code is loaded

CFAR Distillates: Limitations

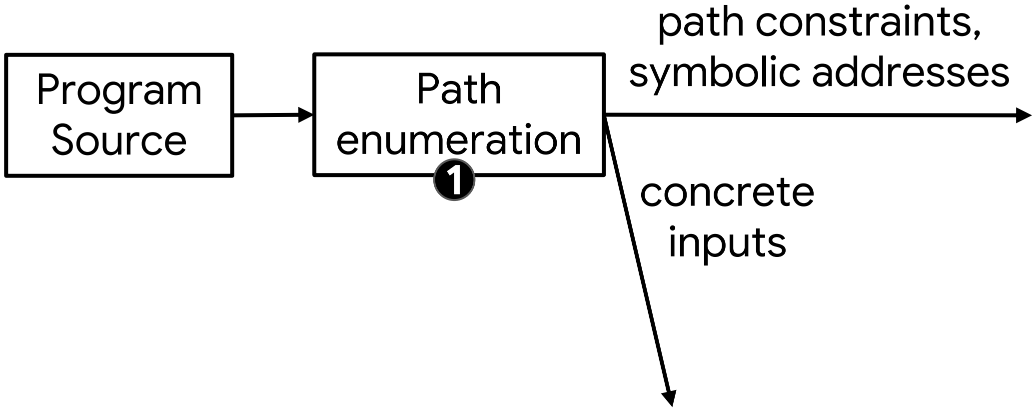
- Discard all timing information
  - Cannot reason about latency
  - Cannot reason about timeliness of prefetch operations

- Does not provide details about speculative memory accesses
  - Hidden by the hardware

CFAR Overview



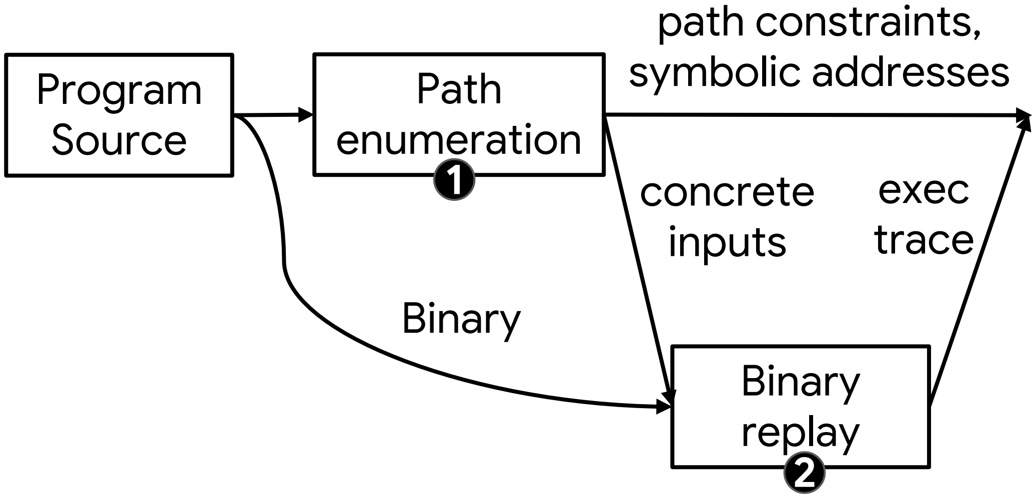
CFAR Distillation: Step 1



- Analyze source to enumerate paths through the program
  - Tradeoff between completeness, scalability, and human effort

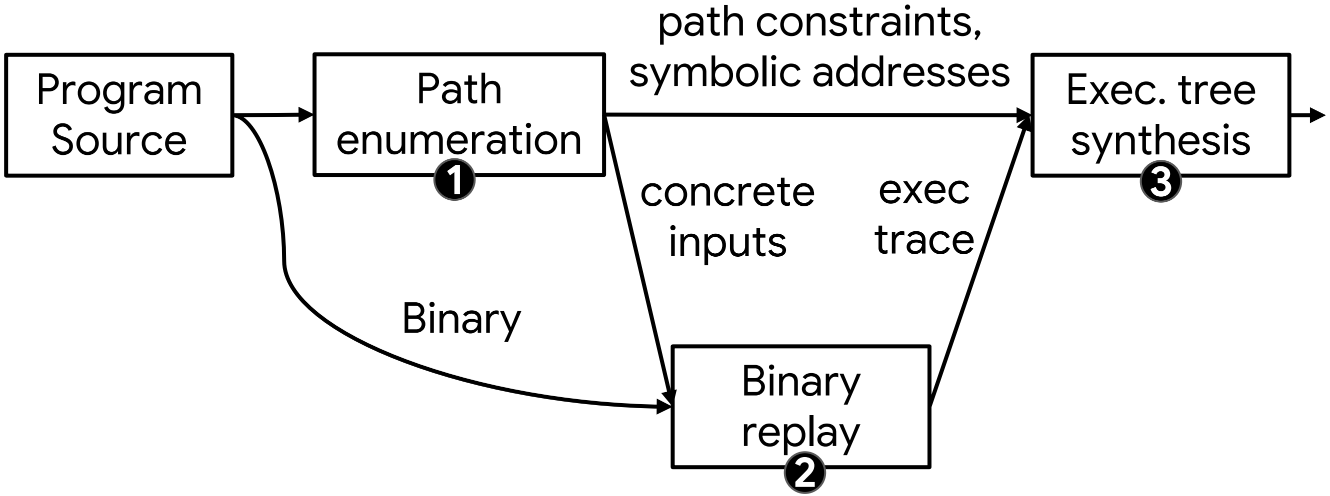
- CFAR currently provides three types of analysis
  - Automated symbolic execution: poor scalability
  - Guided symbolic execution: requires human effort
  - Concolic execution (WIP): incomplete

CFAR Distillation: Step 2



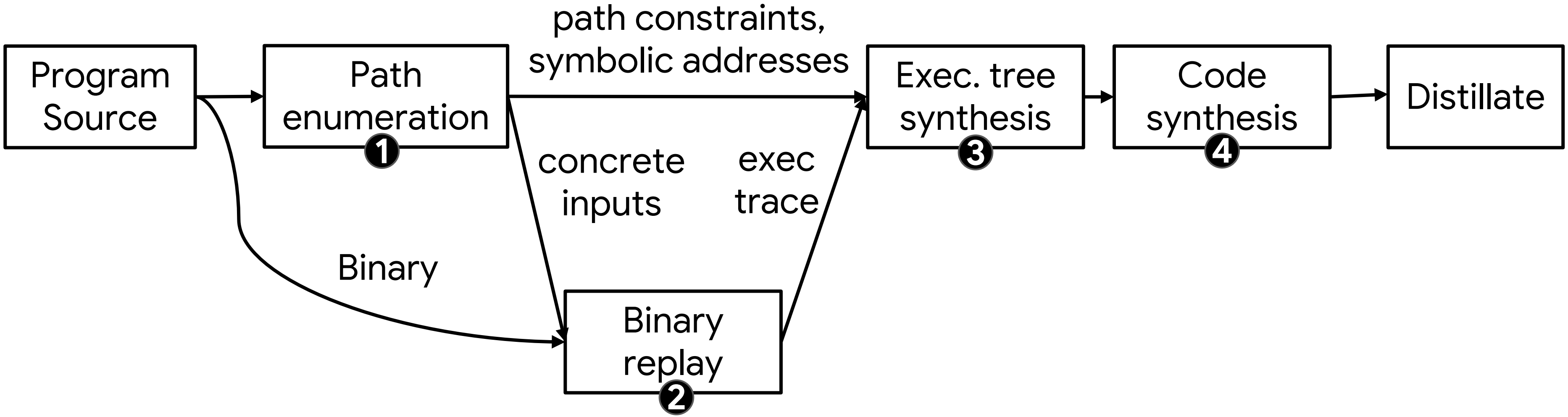
Replay binary to obtain precise mem. access trace for each path

CFAR Distillation: Step 3



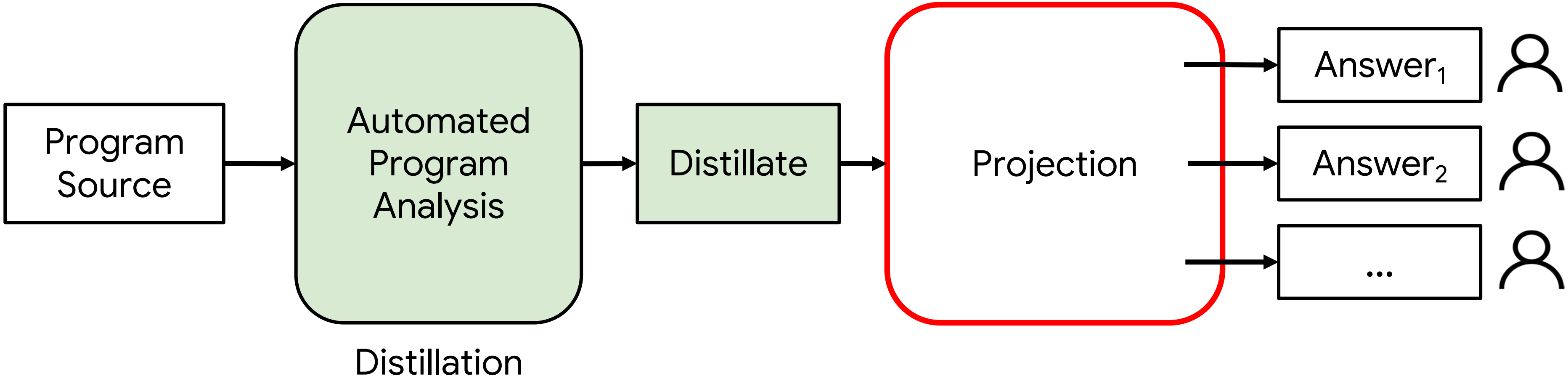
Collate execution trace and symbolic addresses per path
Synthesize execution tree containing all paths using path constraints

CFAR Distillation: Step 4

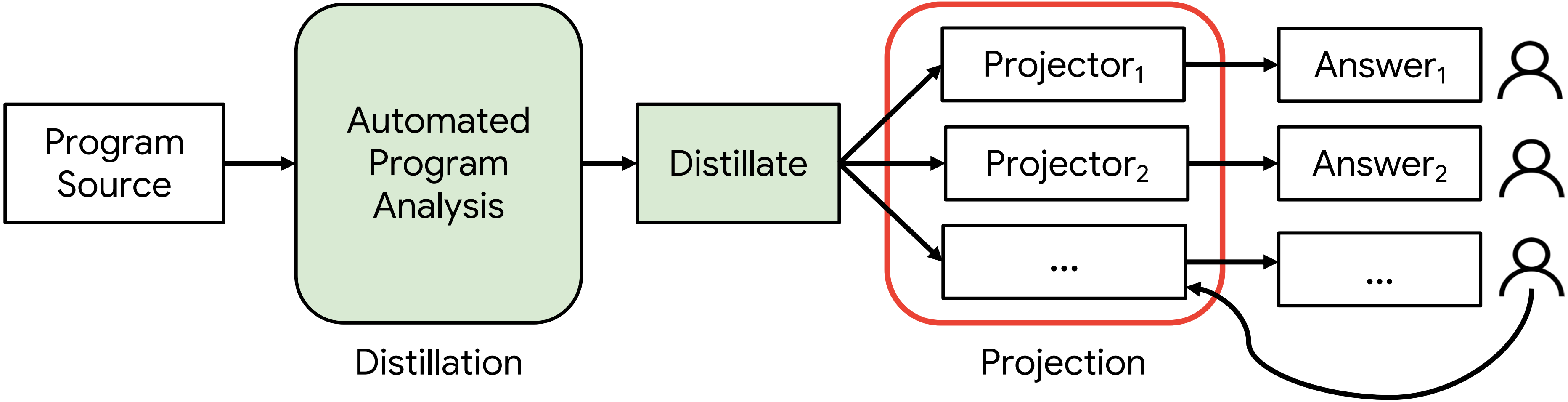


Translate execution tree into Python program for readability

CFAR Overview



CFAR: Projection



CFAR: Projectors

- User-defined functions that compute different cache-usage properties
    - Input: Python list containing symbolic memory accesses
    - Output: Answer to question about cache usage

- For example:
    - `len(list)` returns number of memory accesses
    - `len(set([x.addr//64 for x in list]))` returns unique cache lines touched

CFAR-Provided Projectors

- CFAR comes with three projectors that answer FAQs about cache usage
  - P_{scale}: how cache usage scales as a function of workload
  - $P_{h/m}$: cache model to study hit and miss profile
  - P_{crypto}: identifying secret-dependent branches, memory accesses

CFAR-Provided Projectors

- CFAR comes with three projectors that answer FAQs about cache usage
  - P_{scale}: how cache usage scales as a function of workload
  - $P_{h/m}$: cache model to study hit and miss profile
  - P_{crypto}: identifying secret-dependent branches, memory accesses

- Projectors are easy to write
  - P_{scale} and P_{crypto} are both < 100 lines of Python
  - The cache model in $P_{h/m}$ is largely taken from gem5

Projectors directly operate on lists, are agnostic to how the program being analyzed produced the list

Example Projector: P_{scale}

- Given a list of addresses, and a symbol of interest, compute number of accessed cache lines that will change if the value of the symbol changes
  - E.g., P_{scale} ([500, x+16, x+72], 'x') should return 2

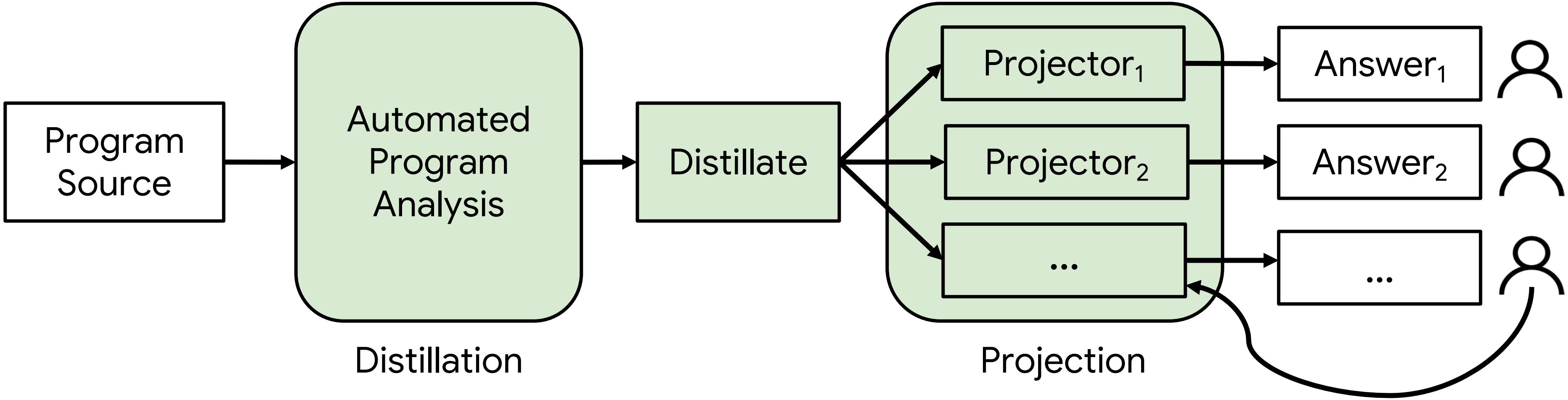
- P_{scale} under the covers: 3 step process
  - Query Z3 to compute list of addresses that may change if x changes
  - Compute concrete values of x for which the change will take place
  - Compute difference in the set of concrete cache lines touched for above values

CFAR Projectors: Limitations

- Analyze each path in isolation
  - Feasible for projectors to analyze >1 list at a time, but CFAR does not support this yet

- Assume program is not preempted during execution
  - Infeasible to analyze all possible concurrently-running programs

CFAR: Projection



CFAR: Evaluation

- Programs analyzed:
  - Fast path of TCP ingress, egress from Linux v6.5 and v6.8
    - Also analyzed fast path of a kernel-bypass stack, lwIP stack
  - 2 open-source hash table implementations
  - 51 syscalls from Hyperkernel
  - 7 algorithms from OpenSSL 3.0

- Eval questions: Are CFAR-extracted distillates
  - Accurate?
  - Useful?

Accuracy of CFAR's Distillate

- Manually wrote test-cases that cover ~50% of paths for each program
- Measured number, addresses of
  - Executed instructions
  - Executed data memory accesses
- Compared to values predicted by distillate
- Observed **ZERO** error

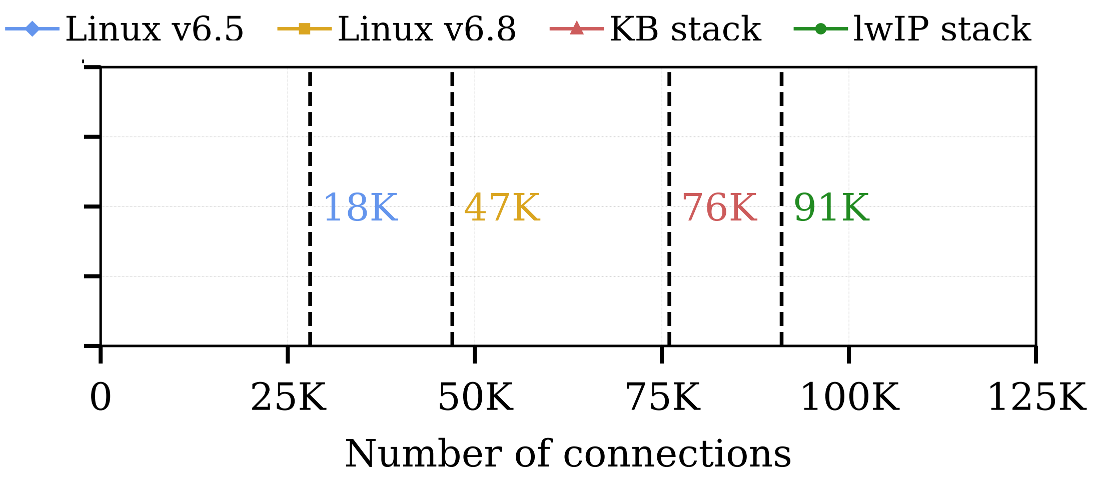
CFAR's distillate is accurate and holds irrespective of concrete values of input/state and address space randomization

How Does Cache Usage Scale?

- Used CFAR to analyze fast path of 4 TCP stacks:
  - Linux before (v6.5) and after (v6.8) recent patch, IX (KB), and lwIP stack
- Predicted number of connections at which each would suffer consistent LLC misses

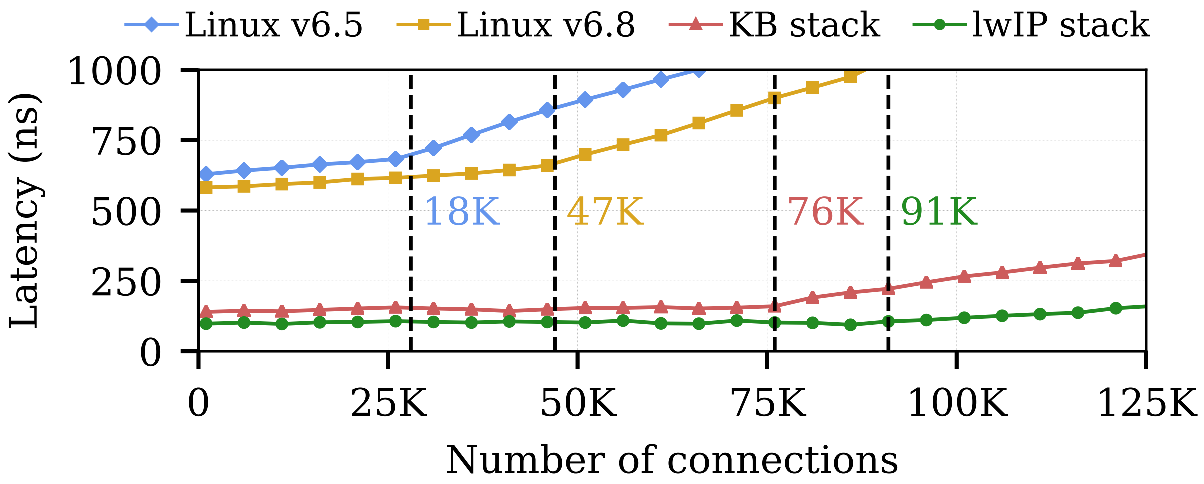
How Does Cache Usage Scale?

- Used CFAR to analyze fast path of 4 TCP stacks:
  - Linux before (v6.5) and after (v6.8) recent patch, IX (KB), and lwIP stack
- Predicted number of connections at which each would suffer consistent LLC misses



How Does Cache Usage Scale?

- Used CFAR to analyze fast path of 4 TCP stacks:
  - Linux before (v6.5) and after (v6.8) recent patch, IX (KB), and lwIP stack
- Predicted number of connections at which each would suffer consistent LLC misses



CFAR can provide developers with clarity into cache usage even for third-party code

Identifying Inefficient Access Patterns

- Identified a case of false sharing in the IX stack using a simple 5 line projector

```
def pcb_offset(seq):
    pcb = sympy.Symbol('pcb')
    # if address is an offset from only the PCB,
    # return (address-PCB)/64
    return [(x-pcb)//64 for x in seq if sympy.
    is_constant(x-pcb)]
```

Identifying Inefficient Access Patterns

- Identified a case of false sharing in the IX stack using a simple 5 line projector

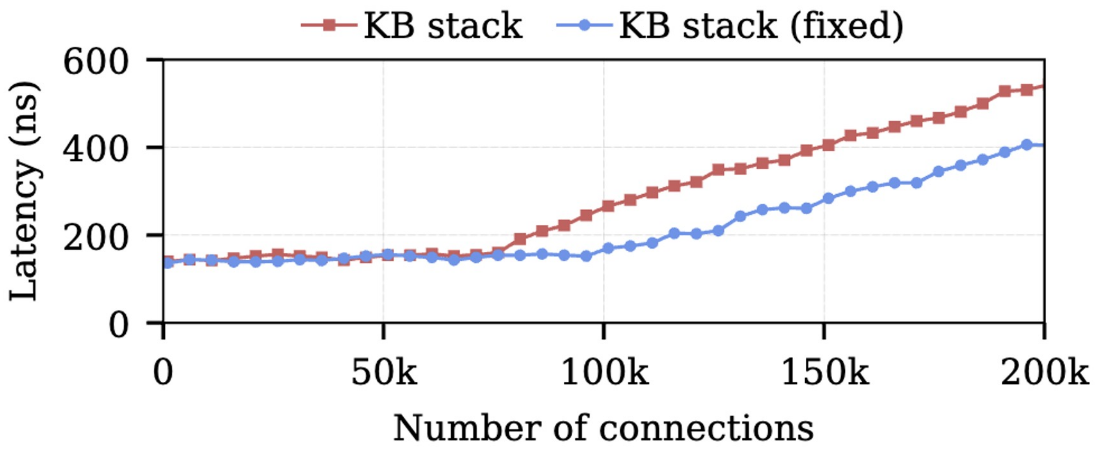
```
def pcb_offset(seq):
    pcb = sympy.Symbol('pcb')
    # if address is an offset from only the PCB,
    # return (address-PCB)/64
    return [(x-pcb)//64 for x in seq if sympy.
    is_constant(x-pcb)]
```

```
# Send fast path: KB stack
# No access to 5th cache line
[2,3,3,1,1,3,3,3,3,1,2,3,2,2,1,1,1,1,0,0,2,1,2,2,1,0,2]
```

```
# Receive fast path: KB stack
# Only one access to 5th cache line
[1,1,0,0,2,2,3,4,1,2,2,3]
```

Identifying Inefficient Access Patterns

- Re-organized `struct tcp_pcb` for cache efficiency (confirmed by projector)



CFAR enables developers to identify inefficient access patterns without elaborate benchmarking

Identifying Cache-Based Leakages

- Inspected 7 algorithms from OpenSSL 3.0 with P_{crypto}
  - AES, SHA, MD5, Poly1305, Chacha, echde, RSA
- Reproduced known cache-leakage vulnerability in RSA (OpenSSL 1.0)
- Found a new constant-time violation in AES, latent since OpenSSL 1.1
  - Acknowledged by maintainers, in final stages of being merged

Since the memory distillate is precise, developers can use CFAR to analyze more than just performance properties of code

Constant-Time Violation in AES

```
def ossl_cipher_unpadblock_icache(buffer, buffer_length, block_size):
```

Constant-Time Violation in AES

Projection showing constant-time violation

```
def ossl_cipher_unpadblock_icache(buffer, buffer_length, block_size):

  if buffer.padding_length == 0:
    return 44
  else:
    if buffer.padding_length > block_size:
      return 48
    else:
      return 57 + 19*buffer.padding_length
```

Constant-Time Violation in AES

Projection showing constant-time violation

```
def ossl_cipher_unpadblock_icache(buffer, buffer_length, block_size):

  if buffer.padding_length == 0:
    return 44
  else:
    if buffer.padding_length > block_size:
      return 48
    else:
      return 57 + 19*buffer.padding_length
```

Constant-Time Violation in AES

Projection showing constant-time violation

```
def ossl_cipher_unpadblock_icache(buffer, buffer_length, block_size):

  if buffer.padding_length == 0:
    return 44
  else:
    if buffer.padding_length > block_size:
      return 48
    else:
      return 57 + 19*buffer.padding_length
```

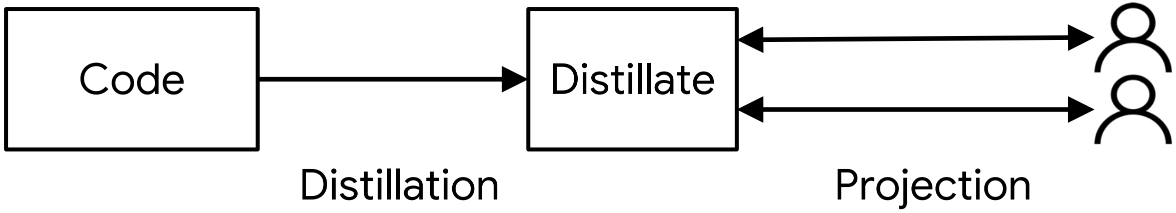
Projection after fix

```
def ossl_cipher_unpadblock_icache(buffer, buffer_length, block_size):

  return 2985
```

Cache Footprint AnalyzeR (CFAR)

- Key idea: abstraction of memory distillate
  - Captures details relevant to how code accesses mem, discards all else
  - Can be projected into answers to diverse questions about cache usage



https://dslab.epfl.ch/research/perf

Backup Slides

Loop Summarization in CFAR

- Does not impact precision, only readability
- Best-effort process
- Uses templates for “common” loop access patterns [DMON OSDI’21]
  - 2 array-based, 2 pointer-chasing patterns
- Requirements:
  - Loop body does not branch on value of iteration counter
  - Maximum of 2 termination conditions for the loop.

Loop Summarization in CFAR: Example

```
def memcmp_dcache(s1,s2,len):

    if Exists(i,And(0<=i<len,[s1+i]!=[s2+i],
                     ForAll(j, Implies(0<=j<i),[s1+j]==[s2+j])))):

      return ForAll(k, Implies(0<=k<=i),[(r,s1+k),(r,s2+k)])
    return ForAll(k, Implies(0<=k<=len),[(r,s1+k),(r,s2+k)])
```