Automatically Reasoning About How Systems Code Uses the CPU Cache

Rishabh Iyer, Katerina Argyraki, George Candea
EPFL, Switzerland

Abstract

We present CFAR, a technique and tool for developers to reason precisely about how their code, as well as third-party code, uses the CPU cache. Given a piece of systems code *P*, CFAR employs program analysis and binary instrumentation to automatically "distill" how *P* accesses memory, and uses "projectors" on top of the extracted distillates to answer specific questions about *P*'s cache usage. CFAR comes with three example projectors that report (1) how *P*'s cache footprint scales across unseen inputs; (2) the cache hits and misses incurred by *P* for each class of inputs; and (3) potential vulnerabilities in cryptographic code caused by secret-dependent cache access patterns.

We use CFAR to analyze a performance-critical subset of four TCP stacks—two versions of the Linux stack, a TCP stack used by recently proposed kernel-bypass OSes, and the lwIP TCP stack for embedded systems—as well as 7 algorithms from the OpenSSL cryptographic library, all 51 system calls of the Hyperkernel, and 2 hash table implementations. We demonstrate how CFAR enables developers to not only identify performance bugs and security vulnerabilities in their own code, but also understand the performance impact of incorporating third-party code into their systems without running elaborate benchmark suites.

CFAR is available at https://dslab.epfl.ch/research/cfar.

1 Introduction

System performance is important, yet it is often poorly understood. For this reason, we recently proposed the notion of *performance interfaces* [29, 30, 46] by analogy to semantic interfaces (e.g., abstract classes, specifications, documentation), which have been used for many decades to succinctly describe a program's functionality. A performance interface describes a system's performance behavior in a manner that is simultaneously succinct, precise, and human-readable. Our goal with performance interfaces is to help developers efficiently reason about the performance behavior of their own, as well as third-party code without having to delve into the code's implementation details; just like semantic interfaces help developers reason about functionality today.

Low-level systems code (e.g., operating systems, device drivers, network stacks, etc) is special, because performance often critically depends on how the code interacts with the underlying micro-architecture. As a result, system developers spend a lot of time trying to understand this interaction, e.g., trying to understand whether the code's memory-access patterns are cache-friendly [2, 12–14, 49, 69], and whether the code's working set fits in cache [18, 22, 44, 66, 67, 76]. *Not* understanding this interaction can lead to performance bugs that are hard to diagnose, and also result in unexpected performance behavior when using third-party code. For instance, a recent patch showed [41] how the fast path of the Linux TCP stack had been incurring a bloated cache footprint for over 10 years leading to slowdowns of up to 45%, and prior work has shown that applications may run up to 4× slower after invoking third-party code (e.g., a syscall) due to the code's micro-architectural footprint [65, 73].

The goal of this work is to help system developers answer key questions about how their code, as well as third-party code, interacts with the underlying micro-architecture. We focus on interactions with the CPU caches (both data and instruction caches), since these often play a critical role in the performance of systems code [12–14, 18, 22, 37, 44, 56, 66, 67, 69, 76, 77]. We seek to answer frequently-asked questions about cache usage such as: "*How does the code's cache usage scale as a function of the workload?*" [6, 18, 22, 66, 67], and "*Which workloads make the code's working set exceed the cache size*?" [37, 56] without requiring developers to delve into the code's details or run elaborate benchmarks.

Answering the above questions requires visibility into how the code processes an *abstract* workload; hence we look for abstractions that capture how the code interacts with the caches as a function of the workload, in a succinct, precise, and human-readable manner. Our approach is in contrast to existing performance-analysis tools such as profilers [10, 42, 57, 68] and cycle-accurate simulators [7, 9]. Such tools can only provide insights into cache usage for the *con-*

crete workloads with which the code is profiled or simulated; they cannot provide visibility into how the code behaves for *abstract* unseen workloads. As a result, when using these tools, developers are forced to manually reverse-engineer the answer to their questions. This process is both time-consuming and error-prone [28], particularly for code that the developers did not write themselves.

We present Cache Footprint AnalyzeR (CFAR): a tool that processes a piece of systems code into answers to developers' questions about how that code uses the cache. CFAR's processing consists of two phases: In the first phase, CFAR takes as input the code and extracts from it an abstract representation (a "distillate") that contains all the information on how the code accesses memory. In the second phase, CFAR uses simple programs ("projectors") to transform the distillate into answer specific questions about the code's cache usage; since the distillate is a precise abstraction of the code's memory usage (i.e., it contains all the information relevant to how the code accesses memory), developers can use projectors to answer diverse questions about the code's cache usage. Under the covers, CFAR relies on a combination of static analysis, symbolic execution, and binary instrumentation to automatically extract distillates. We chose these particular program-analysis techniques because, despite their scalability limitations (discussed in §4.3), they enable precisely the visibility developers seek, namely reasoning about how the code processes an abstract workload.

The current CFAR prototype comes with three projectors that answer frequently-asked questions about cache usage: (1) $\mathcal{P}_{\text{scale}}$ describes how the amount of data the code brings into the cache (i.e., the number of unique cache lines touched) varies as a function of the workload, (2) $\mathcal{P}_{\text{h/m}}$ details whether each memory access will hit or miss in the cache as a function of workload, and finally (3) $\mathcal{P}_{\text{crypt}}$ flags cryptographic code that branches or accesses memory addresses depending on secret inputs, thereby flagging potential branch- and cache-based leakages. $\mathcal{P}_{\text{crypt}}$, in particular, demonstrates the flexibility of CFAR's two-phased process: since the distillate contains all information relevant to how the code accesses memory, developers can write projectors to analyze more than just performance properties. We envision developers contributing more such projectors, making CFAR more useful over time. Eventually, developers will just use whatever ships with CFAR, extending it only when they cannot get the answer they seek.

We use CFAR to analyze a performance-critical subset of the transport layer of 4 TCP stacks—2 versions of Linux stack (i.e., before, and after the recent reorganization for cache efficiency [41]), a TCP stack used by recently proposed kernel-bypass OSes [6], and the lwIP TCP stack for embedded systems [19]—as well as 2 hash table implementations [58, 72], all 51 of the Hyperkernel's system calls [50], and 7 algorithms from the OpenSSL cryptographic library [53]. We use the results to demonstrate how distillates and projectors enable developers to understand the cache usage of their own or third-party code, for unseen workloads, without running elaborate benchmark suites. As part of our evaluation, we also uncovered a cache-inefficient data layout in the kernel-bypass stack, an error path in the Hyperkernel `mmap()` system call that, despite looking innocuous, inadvertently pollutes 40% of the L1 d-cache; and a constant-time violation in OpenSSL 3.0's implementation of AES. For all the above code, CFAR's analysis completes in minutes, which makes us confident that extraction and analysis of distillates can be feasibly integrated into the regular software development cycle.

The rest of this paper is organized as follows: we first motivate CFAR using examples of cache-usage questions that existing tools cannot answer (§2), before providing an overview of the CFAR approach (§3) and detailing its design (§4). We then evaluate CFAR experimentally (§5), discuss related work (§6), and finally conclude (§7).

2 Motivation

In this section, we give an example of the kind of questions that systems developers ask about their code's cache usage (§2.1), and then describe why existing tools cannot answer such questions (§2.2).

2.1 Example

Consider a developer Alice, who is building a fast, in-memory key-value store. The key-value store uses a hash table to store the key-value pairs and runs atop a user-space, kernel-bypass transport stack. Alice has modified an existing hash table implementation to suit her needs and thus understands that part of the code well. However, she is using an off-the-shelf transport stack [19, 33, 74], of which she understands little beyond the semantic interface it exposes.

In such a system, throughput is often bottlenecked by the number of last-level cache (LLC) misses per request [40, 66, 78]; hence, to optimize throughput, Alice needs to know how the different parts of her code use the cache and how they affect the LLC misses as a function of the workload. For example, if her system fails to reach the expected throughput due to persistent LLC misses, what is the predominant cause? Is it that the hash table code touches too many cache lines per `put()` or `get()` request? Or is it that the transport stack's buffer-management code touches too many cache lines per connection [6]? In the former case, Alice should spend her time optimizing the memory layout of the hash table [12–14], whereas in the latter, she should port her code to alternative stacks with smaller memory footprints [19, 66]. Finally, if both codebases were already highly optimized, she should avoid wasting time on code optimizations and replicate her service across machines [4].

2.2 Existing Tools are Insufficient

Existing tools such as profilers [10, 42, 57, 68] and cycle-accurate simulators [7, 9] are fundamentally ill-suited to answering Alice's questions. This is because profilers and simulators are designed to reason about what the code does to the micro-architecture for a *given workload*, whereas answering Alice's questions requires reasoning about what the code does to the micro-architecture as a *function of the workload*. So, while profilers and simulators can provide visibility into the code's cache usage for a given workload, they do not have *predictive power*, and thus cannot provide Alice with visibility into cache usage for workloads beyond the ones that she (herself) provided the tools with.

As a result, developers like Alice are forced to guess the answers to their questions based on (incomplete) information derived from profiling. For example, Alice would typically profile her system with many workloads to measure micro-architectural events and then guess the predominant cause of LLC misses. In particular, she would try to identify the properties of workloads that led to low throughput: were they those that led to a large number of `put()` or `get()` accesses per request? Or those that led to a large number of concurrent connections? This is similar to what developers at Google do to answer such questions, e.g., they run their code for multiple workloads, use profilers to count the total number of unique cache lines touched, and then manually extrapolate how workload affects their code's cache footprint [5, 41].

Reasoning about cache usage in such a manner is not only time consuming but also error prone, particularly for third-party code. For instance, Alice (who knows little about the implementation of the transport stack) may not even think about running workloads that lead to different numbers of concurrent connections. In general, performance profiling suffers from the "large input problem" [47, 52], i.e., the fact that unexpected performance behavior often manifests only when input size (e.g., the number of concurrent connections) exceeds some limit that may seem arbitrary to those who are not intimately familiar with the code. So, designing a test suite that completely covers a system's performance behaviors is hard, and developers do not even have well-defined coverage metrics for the same. For example, while line coverage is used as a proxy for the coverage of semantic behaviors, performance profiling does not even have such an imperfect metric.

As a result, developers like Alice often fail to identify workload properties that significantly impact cache usage, causing performance cliffs to manifest in production. For instance, developers from Google recently demonstrated how the fast path of the Linux TCP stack had been accessing 50% more cache lines than it needed to for over 10 years, which was leading to performance degradations of up to 45% [41]. Similarly, initial work on predicting the working set of network functions ignored the impact of different packet sizes [18], and a study of Linux's syscall performance showed how a newly introduced configuration parameter can destroy spatial locality and lead to increased LLC misses [61]. In practice, developers like Alice often overestimate their system's cache usage and overprovision resources for their services to mitigate unexpected throughput degradation due to incomplete performance profiling [23]; however this leads to lower system efficiency and inflated costs.

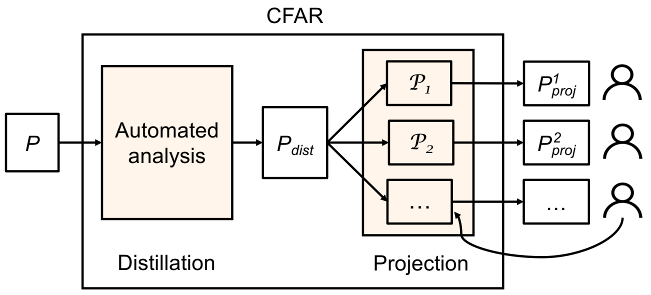

Figure 1: The CFAR workflow. P denotes a piece of systems code, P_{dist} denotes the corresponding distillate, $\mathcal{P}_i$ denotes the different projectors, and P^i_{proj} denotes the corresponding projections that provide answers to developers' questions about P's cache usage.

Summary. Existing tools such as profilers and cycle-accurate simulators are ill-suited to answering frequently-asked questions about cache usage since they do not have *predictive power* across workloads. As a result, developers are forced to estimate the answers to their questions using incomplete information obtained via profiling. This process is not only time consuming but also error prone, particularly for third-party code.

3 CFAR Overview

Since answering questions about cache usage requires reasoning about the code, we look for abstractions that precisely capture what the code does to the cache as a function of the workload.

With this in mind, we propose two abstractions: *distillates* and *projections*. Let P be any well-defined part of a system that can be invoked individually, such as a syscall in an OS kernel or a function in a library, or even a standalone program. A *distillate* P_{dist} is a program that specifies precisely and completely how P accesses memory. A *projection* P^{π}_{proj} is a program that answers a specific question π about P's cache usage. For any given P, there exists a unique distillate P_{dist}, but there can be as many projections as there are questions about P's cache usage.

We represent distillates and projections as programs—as opposed to denser, more mathematical representations (e.g., [21])—for two reasons. First, programs provide developers with a representation that they are familiar with, allowing them to quickly read and understand the answers to questions about cache usage. Second, programs can be executed, which makes it possible for tools to leverage distillates and projections for automated performance analysis; in §4 we show how CFAR executes a distillate against a cache model to reason about cache hits and misses.

Fig. 1 illustrates CFAR's workflow, which consists of two phases: The first phase takes as input a piece of code P and

```
 1 int sys_create(int fd, fn_t fn, uint64
       ftype, uint64 value, uint64 omode) {
 2
 3     if (ftype == FD_NONE)
 4         return -EINVAL;
 5     if (!is_fd_valid(fd))
 6         return -EBADF;
 7     if (&proc_tbl[pid]->ofile[fd] != 0)
 8         return -EINVAL;
 9     if (!is_fn_valid(fn))
10         return -EINVAL;
11
12     struct file = get_file(fn);
13     if (file->refcnt != 0)
14         return -EINVAL;
15     file->type = ftype;
16     file->value = value;
17     file->omode = omode;
18     file->refcnt = file->offset = 0;
19     set_fd(pid, fd, fn);
20     return 0;
21 }
```

```
 1 def sys_create_dcache(fd, fn, ftype, value, omode):
 2     # State: pid, proc_tbl, file_tbl
 3
 4     if ftype == FD_NONE: #6 accesses
 5       return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]
 6
 7     if not(fd >=0 and fd < NOFILE): #6 accesses
 8         return [(w,rsp-8),(w,rsp-16),..,(r,rsp-8)]
 9
10     if [proc_tbl+256*pid+64+8*fd]: #7 accesses
11         return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,(r,rsp-8)]
12
13     if not(fn >=0 and fn < NOFILE): #7 accesses
14         return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd),..,(r,rsp-8)]
15
16     if [file_tbl+40*fn+8]: #9 accesses
17         return [(w,rsp-8),(w,rsp-16),..,(r,proc_tbl+256*pid+64+8*fd)..,(r,file_tbl+40*
       fn+8),..,(r,rsp-8)]
18
19     # Succesful create. 17 accesses
20     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)]
```

Figure 2: Example program on left (Hyperkernel's `sys_create` syscall that creates a new file) and the corresponding data-accesses distillate.

```
 1 def sys_create_icache(fd, fn, ftype, value, omode):
 2     # State: pid, proc_tbl, file_tbl
 3     # sys_create abbreviated as s
 4
 5     if ftype == FD_NONE: #10 insns
 6       return [(r,s),..,(r,s+168),..,(r,s+176)]
 7
 8     #Error paths elided for representation
 9     ......
10
11     # Succesful create. 45 insns
12     return [(r,s),(r,s+8),..,(r,s+160),(r,s+168),(r,s+176)]
```

Figure 3: Instruction-accesses distillate for `sys_create`.

Notation	Description
P	A piece of code that can be invoked individually (e.g., syscall, library functin, program). It takes as input I and has initial state S_0.
Ω	An ordered sequence of memory accesses.
$\mathcal{P}_\pi$	A *projector*. It is a program that defines a function/property $\pi(\Omega)$ related to cache usage.
P_{dist}	The unique *distillate* of P. It is a program that takes as input I and computes Ω as a function of I and S_0, where Ω is P's memory-access sequence.
P^{π}_{proj}	A *projection* of P. It is a program that takes as input I and computes $\pi(\Omega)$ as a function of I and S_0, where Ω is P's memory-access sequence and $\pi(\Omega)$ is defined by a projector $\mathcal{P}_\pi$.

Table 1: Glossary.

automatically extracts P's distillate. The second phase relies on simple programs ("projectors") that transform the distillate into projections that answer specific questions about P's cache usage, such as "*How many unique cache lines does P touch as a function of the workload*?" and "*How does P's cache hit/miss profile varies as a function of the workload*?" CFAR currently provides three such projectors, and we envision developers contributing more over time. Eventually, developers will use whatever ships with CFAR, extending it only when they cannot obtain the answer(s) they seek.

CFAR's two-phased workflow provides flexibility, i.e., enables it to answer diverse questions about cache usage. Since the distillate captures *all* information relevant to how the code accesses memory, it can always be transformed—using a suitable projector—into a projection that answers a specific question about the code's cache usage. We demonstrate this flexibility by building a projector ($\mathcal{P}_{\text{crypt}}$) that goes beyond performance analysis and uses the distillate to identify potential cache-based security vulnerabilities.

CFAR does not make any assumptions about the kind of code that it takes as input. That said, in this work, we focus on systems code (e.g., operating systems, device drivers, network stacks, etc), since it is code for which cache usage has a significant impact on performance.

We now define the three key components of CFAR, namely distillates (§3.1), projectors (§3.2), and projections (§3.3). Table 1 summarizes these definitions.

3.1 Distillates

Consider a program (or function, or method) P, with input(s) I, and state S_0 at the time of invocation. S_0 consists of the values of P's objects in the heap and the stack up to `%esp`.

P's *distillate* P_{dist} is another (simpler) program that takes the same input(s) I, and computes P's sequence of memory accesses Ω as a function of I and S_0. Since accessing data vs. instructions exhibits distinct patterns, we distinguish between a data-accesses distillate P^{data}_{dist} and an instruction-accesses distillate P^{instr}_{dist}. The former computes the sequence of data-memory accesses that would be observed if executing P with input I starting from state S_0, while P^{instr}_{dist} computes the corresponding instruction-memory accesses.

We illustrate what a distillate looks like through the example of the `sys_create` syscall (Fig. 2, left) of the Hyperkernel [50]. First, each memory access in Ω is a tuple $< type, addr >$, where *type* can be a read (`r`), write (`w`) or read-modify-write (`rmw`), while *addr* is a memory address. In a data-accesses distillate (Fig. 2, right), each memory address is a function of standard state components (e.g., the stack pointer `rsp`), as well as components that are specific to P; for example, line 11 in the distillate describes accesses that are a function of `proc_tbl`, `pid`, and `fd`, which arise from executing line 7 in `sys_create`. If a memory address is independent of I and S_0 (e.g., the address of a struct allocated by P in the heap and then freed before returning), it is represented as a named constant (e.g., `mallocRetVal@file.c:342`). In an instruction-accesses distillate (Fig. 3), each memory address is represented as aligned offsets relative to the address of the first instruction in P. In our particular example, the compiler inlines all helper functions, hence there is only one base address `s`.

The distillate P_{dist} is a precise and complete representation of P's memory usage. It is *precise* because it correctly predicts

```
def sys_create_dcache_num_accesses(fd, fn, ftype, value, omode):
    # State: pid, proc_tbl, file_tbl

    if ftype == FD_NONE:
      return 6

    if not(fd >=0 and fd < NOFILE):
        return 6

    if [proc_tbl+256*pid+64+8*fd]:
        return 7

    if not(fn >=0 and fn < NOFILE):
        return 7

    if [file_tbl+40*fn+8]:
        return 9

    # Succesful create.
    return 17
```

Figure 4: Projection of `sys_create` that describes the number of data memory accesses.

the sequence of memory accesses for any execution of P. The symbolic expressions for data- and instruction-memory accesses as a function of I and S_0 are precise by construction, and therefore correct for any concrete instantiation of I and S_0. The distillate is *complete* in that it contains all information on P's memory accesses that can be found in P. No matter what the concrete values of I and S_0, how the address space is randomized [1], or where in memory the code is loaded, a distillate will always be able to produce the exact sequence of memory accesses that P makes when executing from S_0 with input I.

3.2 Projectors

A *projector* $\mathcal{P}_\pi$ is a program that defines a function π related to cache usage. For example, a projector may define $\pi(\Omega) = |\Omega|$, i.e., the number of memory accesses in Ω, while another projector may define $\pi(\Omega) = |\{\lambda(r) = r/64 : r \in \Omega\}|$, i.e., the number of unique 64-byte cache lines accessed as part of Ω.

We think of a function π as defining a question about cache usage (e.g., "*How many memory accesses does does this piece of code perform?*" or "*How many unique cache lines does the code access?*"). CFAR enables developers to write their own projectors, such that they can formulate their own questions.

A key property of projectors is that they are *code-agnostic*; i.e., $\mathcal{P}_\pi$ defines a function of Ω, without taking into account the semantics of the code that produced Ω. This code-agnostic nature makes projectors easy to express and enables developers to inspect the cache behavior of third-party code without having to delve into its implementation details.

A function π may take inputs beyond just Ω. For example, $\pi(\Omega, \$M)$ may specify the number of hits and misses incurred in the L1 data cache given a particular cache model $\$M$. Since Ω is independent of where and how the code that produced it was executed, such a generalized function π can precisely characterize the impact of running a piece of code on different micro-architectures and with different OS configurations (e.g., the page sizes that P is backed by).

3.3 Projections

A *projection* P^{π}_{proj} of P is another (simpler) program that, just like P, takes as input I, and computes the value of function π for P, as a function of I and S_0. Said differently, if we think of π as a specific question about cache usage, then P^{π}_{proj} is a program that provides the answer to that question for P. Fig. 4 shows a projection of `sys_create` that computes the number of data memory accesses performed by the system call as a function of its input and the OS state. CFAR produces P^{π}_{proj} by applying the projector $\mathcal{P}_\pi$ to P's distillate P_{dist}.

Summary. CFAR provides abstractions—distillates and projections—that precisely capture what a piece of code does to the micro-architecture as a function of its inputs; and a simple means—projectors—to query these abstractions. By combining these elements, CFAR enables developers to efficiently answer diverse questions about the cache usage of systems code without having to delve into the implementation details of the code or run elaborate benchmarks.

4 CFAR Design

We now describe how the two phases of CFAR's processing, namely distillation (§4.1) and projection (§4.2), work.

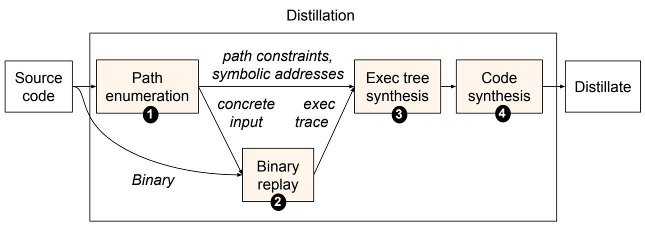

Figure 5: The four steps in CFAR's distillation process.

4.1 Phase #1: Distillation

CFAR automatically distills an input program (P) into its corresponding distillate (P_{dist}) using a four step process (shown in Fig. 5): it (i) enumerates all feasible executions paths in P using automated program analysis; then (ii) obtains a binary execution trace for each such path; then (iii) based on these two outputs, prepares an execution tree for the distillate; and lastly (iv) optimizes this tree and produces P_{dist}.

While the use of automated program analysis and binary replay ensures that CFAR can extract a precise distillate without requiring any effort on the part of the developer, it does come with limitations. Most notably, CFAR is subject to the scalability limitations of automated program analysis and is thus a poor fit for multi-threaded code, and code with loops for which bounds on the number of iterations cannot be statically computed. Additionally, CFAR is also limited by the proprietary nature of modern hardware. For instance, since the exact algorithms used to schedule instructions in an out-of-order processor are not publicly available, CFAR cannot reason about speculative memory accesses. We describe CFAR's limitations in detail in §4.3, but emphasize that despite its limitations, CFAR can provide useful information about cache usage for a wide variety of systems code (§5).

4.1.1 Step #1: Path enumeration

To obtain all the paths in P, CFAR uses *exhaustive* symbolic execution to enumerate them. Symbolic execution [11, 25, 35, 64] is a program analysis technique that automatically traverses the feasible execution paths of a body of code, enabling a comprehensive analysis of its control flow. The technique is powerful, but also faces challenges related to loops and pointers, which we discuss in §4.3. We use an exhaustive form of this technique, which yields *all* feasible paths in P.

For each enumerated path α, CFAR saves four key pieces of information: (1) the precise path constraint C_α that uniquely defines this path, i.e., the conjunction of the outcomes of evaluating each `if` predicate along α, (2) a concrete input I_α that exercises this path, obtained by asking an SMT constraint solver for a satisfying assignment to the input variables in C_α, (3) the symbolic expression corresponding to the address of each data/instruction memory location accessed along the path, expressed as a function of P's inputs and/or state, and finally (4) a corresponding `filename:linenum` identifier for each memory operation, to be used later. The sequence of these symbolic expressions is ω_α.

Our CFAR prototype uses KLEE [11] to perform the symbolic execution. KLEE, like many other SE engines, analyzes the code at the IR level, which in KLEE's case is LLVM [15]. Our KLEE modifications and additions total approximately 1,500 lines of C++.

4.1.2 Step #2: Binary replay

What actually executes on the hardware is not the source code or the IR. Compiler optimizations, such as link-time optimization, cause the executing machine code to not directly correspond to what is in the IR. Furthermore, many IRs are Static Single Assignment (SSA), in which each variable is assigned exactly once. This makes the data flow and dependencies among variables more explicit and easier for the compiler to analyze, but also implies an infinite register file. However, processors do not have infinite register files, so during an actual execution register values often need to be spilled to the stack. Since these pushes and pop to the stack are not captured when analyzing P in the IR form, the corresponding memory accesses will not appear in ω_α.

Therefore, CFAR replays an *instrumented* version of the P binary for each I_α, to obtain the corresponding concrete execution trace X_α. For each machine instruction executed in X_α, CFAR saves: (1) the program counter, (2) the instruction opcode, such as `push` or `pop`, (3) the concrete memory addresses accessed, and (4) the corresponding `filename:linenum` debug information inserted into the binary by the instrumentation.

We deliberately split the analysis into a source-based and a binary-based step in CFAR. It is easier to extract symbolic expressions for memory operations by analyzing the source or the IR. On the other hand, analyzing the binary enables CFAR to be fully precise with respect to compiler optimizations and which instructions lead to memory accesses and do not merely manipulate CPU registers. In theory, these two steps could be combined into a single one by directly symbolically executing the binary. To answer with certainty, one would need to assess how CFAR is affected by the loss of type information when going from source code to binaries.

The CFAR prototype uses Intel PIN [45] to instrument binaries. Our Pintool consists of approximately 350 lines of C++.

4.1.3 Step #3: Synthesizing the distillate's execution tree

In this step, CFAR combines the information extracted in the previous two steps. For each path α in P, it combines the symbolic memory trace ω_α with the corresponding binary execution trace X_α. To produce a *data-access trace*, CFAR takes the sequence of concrete addresses from X_α and replaces (using debug information) all input- and state-dependent accesses with the corresponding symbolic expressions from ω_α, thus producing Ω_α^{data}. To produce an *instruction-access trace*, CFAR uses the program counter values and the call stack in X_α to compute the symbolic offset of each instruction from the start of P (e.g., its entry point, if it is a function or a syscall) and produce Ω_α^{instr}. The call stack gives CFAR information on which function the instruction belongs to, so that it can compute the function-specific offset.

Next, CFAR assembles an execution tree using the path constraints C_α. It arranges all the paths into a tree based on their common prefixes; for every path α there exists a path from tree to leaf in the tree, and vice versa. Each internal node n in the tree contains the predicate corresponding to the original branch in P. The conjunction of the predicates for all internal nodes along a root-to-leaf path forms the corresponding path constraint C_α.

4.1.4 Step #4: Producing the final distillate

The final step in CFAR's distillation process consists of best-effort summarization of loop-related memory access patterns and other, more minor, improvements for human readability of P_{dist}. This step does not elide or lose any information contained in the distillate. It only optimizes the distillate's control flow for human readability since any projection derived from the distillate retains its control flow.

Symbolic execution, by default, unrolls loops and thus produces a different execution path for each loop iteration. This leads to bloated distillates that contain redundant information and are unnecessarily hard to read, particularly if the access pattern of the code does not change across loop iterations.

While automatically summarizing loops in general is undecidable [24], studies have shown that there exist four common categories of loops that relate to data locality issues in systems code [34]. Therefore, CFAR contains loop-summary templates for these four categories of loops—two that traverse array-like data structures, and two that traverse pointer-chasing data structures (e.g., linked lists, trees). All four categories require the loop body to not branch on the precise value of the iteration counter, and for the loop to have a maximum of two termination predicates, one in the loop definition and at most one `break` in the body. In case CFAR is unable to infer that these templates hold, the distillate presents the unrolled loop.

After performing best-effort loop summarization, CFAR transforms the optimized tree into a program that represents P_{dist}. This program takes the same input as P. Every internal tree node n leads to an `if` statement in the program, branching on the predicate contained in that node. Each path through the program ends with a `return` of the corresponding Ω_α—depending on the memory-type of the distillate, this is either Ω_α^{instr} or Ω_α^{data}.

Our CFAR prototype uses Python to represent distillates, because it is one of the most widely used languages [51] and has an easy-to-understand syntax.

Fig. 6 illustrates a snippet for `memcmp`'s P_{dist}^{data} distillate which contains an example of CFAR's loop-summarization optimization. The corresponding loop belongs to the first category mentioned above. Our CFAR prototype uses first-order logic to summarize loops with primitives from Z3's Python API [71]. The predicate that starts on line 3 identifies the smallest index `i` (bounded by `len`) at which the two strings differ. The distillate then states that the memory accessed corresponds to every element of the two arrays up to `i`.

```
1 def memcmp_dcache(s1,s2,len):
2
3     if Exists(i,And(0<=i<len,[s1+i]!=[s2+i],
4                         ForAll(j, Implies(0<=j<i),[s1+j]==[s2+j]))):
5
6       return ForAll(k, Implies(0<=k<=i),[(r,s1+k),(r,s2+k)])
7     return ForAll(k, Implies(0<=k<=len),[(r,s1+k),(r,s2+k)])
```

Figure 6: P_{dist}^{data} for `memcmp` showing CFAR's loop summarization.

4.2 Phase #2: Projection

The distillate produced by the previous phase is a precise and complete description of P's memory-access behavior. While the answers to developers' cache-usage questions can be found in the distillate, they are buried in details that may not be relevant to the specific question being asked. The projection phase turns distillates into actual, clear answers.

4.2.1 Defining Projectors

As described in §3, CFAR enables developers to define projectors: programs that take as input Ω (as a Python list [59] in our prototype) and define specific properties $\pi(\Omega)$ related to cache usage. Given a projector $\mathcal{P}_\pi$ and a distillate P_{dist}, CFAR outputs a program that we call a projection (P_{proj}^{π}). P_{proj}^{π} has the same control flow as P_{dist}, but returns $\pi(\Omega)$ instead of Ω.

CFAR comes with three example projectors: (1) $\mathcal{P}_{\text{scale}}$, which computes how the cache footprint scales across an entire range of previously unseen inputs (e.g., how it varies with the number of active network connections), (2) $\mathcal{P}_{\text{h/m}}$, which computes the cache hit and miss profiles per class of input as opposed to per specific, concrete input, and (3) $\mathcal{P}_{\text{crypt}}$, which flags cryptographic code that accesses the cache in a way that depends on secret inputs; this can be used to flag potential vulnerabilities.

While the above projectors may sound complex, they are fairly straightforward to define since projectors are code agnostic (§3.2). For example, we were able to express the functionality of both $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{crypt}}$ of < 100 lines of Python and while $\mathcal{P}_{\text{h/m}}$ required of approximately 800 lines of Python, almost 600 of those lines were a translation of the cache model in the `gem5` cycle-accurate simulator [7]. In §5.2, we demonstrate how a simple 5-line projector helped us identify a performance bug in a TCP stack used by recently proposed kernel-bypass OSes [6].

4.2.2 CFAR-provided projectors

We now describe the three projectors that our current CFAR prototype provides.

$\mathcal{P}_{\text{scale}}$**:** To compute how the data cache footprint scales for each input to P, $\mathcal{P}_{\text{scale}}$ first determines the number of symbolic addresses in Ω that would differ if only the value of that input changed. It then uses a solver query to check the alignment of those bytes and determine the number of unique bytes touched by these addresses. It presents the results to the user as human-readable formulae. For instance, applying $\mathcal{P}_{\text{scale}}$ to `sys_create` yields the formula: `8*fd + 32*fn` since `sys_create` accesses 8 bytes whose address depends only on `fd` and 32 bytes whose address depends only on `fn` (see line 20). These formulae provide precisely the information that Alice wanted to know for keys and connections in §2. Developers like Alice can thus use $\mathcal{P}_{\text{scale}}$ to quickly estimate when their working set overflows the cache without having to run elaborate benchmarks.

$\mathcal{P}_{\text{h/m}}$**:** $\mathcal{P}_{\text{h/m}}$ allows developers to go a step further and reason about the possible cache misses that their code might incur. $\mathcal{P}_{\text{h/m}}$ takes in four inputs: a trace of memory accesses Ω, an input set size, a cache model and a probability distribution across inputs. The input set size refers to the number of unique inputs (e.g., number of active connections) that the program expects to receive and is used to warm up the cache. The cache model allows developers to specify the microarchitectural characteristics of the hardware that the program is running on; by default $\mathcal{P}_{\text{h/m}}$ provides a 3-level inclusive cache with a next-line prefetcher whose size and set associativity of each level is configurable. Finally, the probability distribution allows developers to specify the relative frequency of different inputs to the program; by default $\mathcal{P}_{\text{h/m}}$ assumes a uniform random distribution.

$\mathcal{P}_{\text{h/m}}$ reasons about possible hits and misses as follows: it first passes the input set size to $\mathcal{P}_{\text{scale}}$, and obtains the total number of cache lines it must account for. Next, it runs the memory trace to insert symbolic addresses corresponding to the input set size into the cache according to the given probability distribution. Finally, it runs the memory trace with symbols corresponding to a random input from the input set and measures the number of hits and misses incurred. To ensure that the effects of the random selection are properly accounted for, $\mathcal{P}_{\text{h/m}}$ repeats the last step multiple times until the set of possible misses stabilizes.

Thus, $\mathcal{P}_{\text{h/m}}$ can be thought of as a symbolic, trace-based cache simulator that allows developers to study cache events without having to write concrete benchmarks to initialize cache state. The only difference is that $\mathcal{P}_{\text{h/m}}$ is forced to deal with symbolic addresses, and so cannot always accurately compute set-associativity conflicts in the cache. Instead, $\mathcal{P}_{\text{h/m}}$ *approximates* set-associativity conflicts by allocating unconstrained, symbolic memory addresses to a random set in the cache, and then mapping all addresses at constant offsets from that symbol based on their least-significant bits of the offset. For example if the address γ is randomly mapped to set μ, then the address $\gamma + 64$ will be deterministically mapped to set $(\mu + 1)$ *mod n*, where n is the total number of sets in the cache. In our evaluation (§5), we show how this approximation works reasonably well in comparison to real hardware.

$\mathcal{P}_{\text{crypt}}$**:** Since the P_{dist} contains all information relevant to how P accesses memory, developers can write projectors that can help reason about more than just performance properties. One such property of interest is the absence secret-dependent branch instructions and data accesses to secret-dependent memory addresses since both of the above are a known source of side channels [3].

$\mathcal{P}_{\text{crypt}}$ takes in three inputs: a trace of memory accesses Ω, the sequence of constraints C that cause the program being analyzed to execute Ω, and a list of program inputs that are secrets. It then uses a Z3 [17] solver query to determine whether the constraints (program branches) are functions of secrets; and (b) the memory addresses are influenced by the secret inputs. In situations where this is the case, $\mathcal{P}_{\text{crypt}}$ returns debug information `filename:linenum` corresponding to the branch/memory-access as well as the path constraints under which it occurs. Note, $\mathcal{P}_{\text{crypt}}$ does not account for all cache-based leakages, it only accounts for leaks due to secret-dependent branches or memory accesses. For example, leakages due to speculatively executed instructions [43] are out of scope for $\mathcal{P}_{\text{crypt}}$.

4.3 Limitations and Assumptions

Scalability limitations of symbolic execution: CFAR's reliance on symbolic execution (SE) makes it subject to SE's own limitations. Depending on which SE engine is used, certain kinds of loops, or symbolic pointers, or multi-threading could prevent obtaining all execution paths [8]. However, there is active research on this topic, and recent SE engines have brought various enhancements that overcome these challenges, such as state merging [38], loop-extended symbolic execution [63], loop summaries [26, 70], loop invariants [32], and symbolic abstract transformers [36].

A CFAR prototype will ultimately be as powerful as its underlying SE engine. Since our prototype relies on KLEE, code whose loops do not have statically computable bounds, or that is multi-threaded, or that has arbitrary symbolic pointers is not an ideal match because path exploration may take too long. This makes our current prototype a poor fit for analyzing entire applications (e.g., key-value stores such as Memcached and Redis). Nevertheless, we believe that CFAR can extract useful distillates for key components even in complex applications (e.g., data structures whose cache footprint is a common source of concern). In our evaluation, we show how developers can use CFAR to precisely reason about the cache usage of two hash table implementations.

Using tool for code not amenable to symbolic execution: Given that automatically extracting complete distillates for arbitrary code is infeasible, we discuss how developers can obtain useful results with CFAR even for such code.

One way to use CFAR for code that cannot be exhaustively symbolically executed is by *constraining the input space*. A reasonable approach is to constrain CFAR to inputs that trigger the "fast path" through the code, since that is a common object of performance analysis. For instance, if the target code is an IP forwarding function, it is reasonable to constrain the distillate to packets without IP options; this dramatically reduces both the size of the distillate and the time required to extract it (since it eliminates the part of the code that loops through the variable-length IP options), while still yielding practically useful results (since performance-sensitive traffic does not typically carry IP options). Focusing on the cache usage of the fast path is common practice today; for instance, the recent re-organization of the Linux TCP stack was based entirely on the requirements of the TCP fast path [41].

CFAR provides a similar interface to KLEE (the SE engine it is based on) for developers to constrain the input space [11]. This interface is easy to use and enables users to provide constraints on arbitrary program variables. In our evaluation, we use this interface when analyzing code that is not amenable to exhaustive symbolic execution (e.g., the Linux TCP stack), and demonstrate that the results can be useful despite the constrained input space.

Constraining the input space requires the user to have some knowledge of the code that is being analyzed (e.g., what typical/fast-path inputs look like); however, we believe this is reasonable since the user likely wants to use the code being analyzed and thus has the required knowledge. We emphasize that constraining the input space does not require users to possess any knowledge of the code's internal implementation details, these are explored automatically by CFAR.

Nevertheless, in scenarios where such knowledge of the code being analyzed is not available, developers can run CFAR while specifying a time limit. When the time limit expires, CFAR outputs a partial distillate that returns the exact sequence of memory accesses performed by the code along the explored execution paths. While this approach is fully automated, the downside is that CFAR may not explore meaningful execution paths in the given time budget. As a result, while our CFAR prototype offers this capability, we did not use it for any of the results shown in our evaluation.

Limitations due to proprietary hardware details: CFAR employs binary instrumentation to obtain an execution trace. Unfortunately, such instrumentation can only reveal instructions that the processor finished executing (retired); it does not reveal instructions that were executed as a result of incorrect speculation (e.g., a mispredicted branch). Such instructions nevertheless could impact the cache and, since CFAR does not see them, the answers computed by projectors may not be fully accurate. We are not aware of any tool that can precisely report such mis-speculated instructions during an execution since the scheduling algorithms used in out-of-order pipelines are proprietary.

Assumptions regarding preemption: In a similar vein, CFAR assumes that P is small enough to not have its execution interrupted by preemption. In the absence of this assumption, the distillate produced by CFAR (i.e., P_{dist}) is still correct, but the projection might not be, because it could be missing third-party cache accesses that occurred during the preemption. In other words, in the presence of preemption, when a projector looks at the symbolic memory trace, it may not get a fully accurate picture of all cache accesses since P was not the only code that ran on the micro-architecture during that period.

5 Evaluation

In this section, we evaluate the CFAR prototype by answering two main questions:

- **Does CFAR work?** We show that CFAR extracts 100%-accurate data- and instruction-accesses distillates, and that this extraction completes in minutes for various kinds of systems code (§5.1).
- **Is CFAR useful** to system developers? We describe four use cases that demonstrate how CFAR provides developers with visibility into cache usage in a way that profilers and simulators cannot (§5.2).

Evaluated programs. We used CFAR to analyze the fast path of the transport layer of 4 TCP stacks—2 versions of Linux stack (i.e., before, and after the recent reorganization for cache efficiency [41]), a TCP stack used by recently proposed kernel-bypass OSes [6], and the lwIP TCP stack for embedded systems [19]—as well as 2 hash table implementations [58, 72], all 51 of the Hyperkernel's system calls [50], and 7 algorithms from OpenSSL 3.0.0 [53]. For the Linux TCP stack, we analyzed the stable versions before and after the reorganization (`v6.5` and `v6.8`), while for all other code, we analyzed the latest stable version. Note, the kernel-bypass stack uses the lwIP stack as a starting point, but heavily modifies the internal data structures and timer management [6].

Put together, these programs challenge CFAR's automated program analysis approach to different extents. At one end of the spectrum are the Hyperkernel and OpenSSL, both of which are amenable to automated program analysis. The hash table implementations occupy the middle of the spectrum, since they are both amenable to *manual* (not automated) program analysis. Finally, at the other end are the four transport layer implementations, which were not written to be amenable to any form of program analysis.

Despite these challenges, we demonstrate that CFAR can provide actionable information about cache usage for each of these programs, by constraining the input space (as discussed in §4.3). To analyze the transport layer of the four TCP stacks, we constrained CFAR to only explore execution paths corresponding to packets processed in the TCP fast path, i.e., packets that belong to an established TCP connection, are received in order, and do not suffer hash collisions with packets from other connections. We picked this particular packet class because it represents a large fraction of packets processed by TCP stack and is the path for which performance matters the most; for instance, the recent re-organization of the Linux TCP stack was based entirely on the requirements of this TCP fast path [41]. To analyze the hash tables, we had to fix the maximum capacity of the table to a concrete value (we picked 65536)—so, our conclusions about this code hold only for this maximum capacity. Given that the implementations take in the capacity as a configurable parameter, we are certain that the memory access pattern is independent of the capacity; we just cannot prove it using symbolic execution.

Setup. We ran all our analyses on an Intel Xeon E5-2690 v2 CPU, clocked at 3.30GHz and provisioned with 25.6MB of LLC and 252GB of DRAM.

5.1 Does CFAR work?

There are two key aspects to determining whether a tool like CFAR works and is practical: does it obtain an accurate abstract representation of performance from the code (§5.1.1), and does it do so in reasonable time (§5.1.2).

5.1.1 Accuracy of distillates

To measure the accuracy of our prototype's distillates, we randomly picked 50% of the execution paths of each program, constructed inputs that exercised each path, counted the number of instructions and memory accesses executed while running each program with each input, and then compared this number to the one predicted by the program's data- and instruction-accesses distillates. In particular, the number of instructions counted during execution should be equal to the number of memory accesses predicted by the instruction-accesses distillate (for the given input), while the number of memory accesses counted during execution should be equal to the number of memory accesses predicted by the data-accesses distillate.

The error was always **zero**, across all programs and inputs, i.e., CFAR's distillates correctly predict every single instruction and memory access executed by the code. This is not surprising since CFAR does not rely on models to predict the sequence of instructions and memory accesses, but instead measures them by replaying the binary. Since CFAR does not modify the program binary in any way, the value measured during replay is identical to the one measured in production.

5.1.2 Time to extract distillates

Table 2 lists how long our prototype takes to extract distillates: for all programs, the analysis completes within 30 mins. The programs that take longest (and the only ones that take more than 15 min) are the Vigor hash table and the `echde` key-generation algorithm in OpenSSL; this is because in both, symbolic execution needed to unroll long loops that iterate over the data structure and compute co-prime numbers, respectively. For all programs, the binary replay, execution tree synthesis, and code synthesis take approximately 2-3 mins in total. The dominant component of CFAR's analysis time—and the one that varies across programs—is symbolic execution.

Program	**Extraction time (mins)**
Linux TCP ingress	11
Linux TCP egress	14
Kernel-bypass TCP ingress	5
Kernel-bypass TCP egress	7
lwIP TCP ingress	4
lwIP TCP egress	5
Hyperkernel syscalls (51 total)	Avg: 4, Max: 7
OpenSSL primitives (7 total)	Avg: 9, Max: 22
Vigor hash table	28
Klint hash table	12

Table 2: Time taken by CFAR to extract distillates.

5.2 Is CFAR useful for system developers?

We demonstrate CFAR's usefulness by presenting four use cases of CFAR answering important questions that developers cannot readily answer with the state of the art: How does the code's working set change with workload (§5.2.1)? How does the layout of data structures in third-party code interact with my code (§5.2.2)? Does my code lead to inefficient memory access patterns (§5.2.3)? Can I prove/disprove the absence of secret-dependent memory accesses (§5.2.4)?

5.2.1 How does the working set change with workload?

We used the $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ projectors to analyze the cache usage of the fast path of the transport layer of the 4 TCP stacks. Recall that this is precisely the question that Alice wanted to answer in §2, but could not do so since profilers and simulators do not provide visibility into how the cache usage scales as a function of the workload.

First, we used $\mathcal{P}_{\text{scale}}$ to predict the number of unique cache lines touched by the TCP fast path assuming symbolic packet contents. The answer was 4, 5, 8, and 12 unique cache lines for the lwIP, kernel-bypass, Linux stack `v6.8` and Linux stack `v6.5`, respectively. We emphasize that while $\mathcal{P}_{\text{scale}}$ provides this information automatically, developers like Alice would either have to run elaborate benchmarks and/or painstakingly pore through the code to obtain this information since it cannot even be gleaned merely by observing the size of the connection-specific `struct`. For example, in Linux the `struct tcp_sock` occupies 42 cache lines in total, but only a fraction of them are accessed on the fast path.

We then passed this information to $\mathcal{P}_{\text{h/m}}$ and asked it to predict when incoming packets were likely to suffer persistent cache misses due to the working set overflowing the LLC (25.6MB on our machine). The answer was that this would occur at approximately $91k$, $76k$, $47k$, and $28k$ concurrent connections for the lwIP, kernel-bypass, Linux stack `v6.8` and Linux stack `v6.5`, respectively. The slight differences in these predictions as compared to simple capacity-based calculations (e.g., $25.6\text{M}/(64 * 4) = 100\text{k}$ connections for lwIP) is due to $\mathcal{P}_{\text{h/m}}$ taking into account conflict misses in addition to capacity misses.

To verify these predictions, we ran a set of experiments where the transport layer receives and sends packets from/to a fixed set of established connections, and we varied the number of connections across experiments. To isolate just the transport layer (which is the code we analyzed), we wrote simple shims for the application and IP layers ourselves. In each experiment, we measured the average latency incurred by packets within the transport layer.

Fig. 7 plots latency as a function of the number of connections. We see that, for each of the three stacks, there is a clear shift around the number of connections predicted by $\mathcal{P}_{\text{h/m}}$. For instance, the latency for the Linux stack `v6.5` increases by only 64ns from 1k to 26k connections, but increases by 211ns from 26k to 52k connections. Likewise—although less visible in the graph due to Linux's dominant latency—the latency for the lwIP stack increases by only 13ns from 1k to 86k connections, it increases by 50ns from 86k to 125k connections. Note, the shift does not occur exactly at the predicted number of connections, but very close to it; compared to the predicted values of 28k,47k, 76k, and 91k, we observed the shifts at 26k, 44k, 72k, and 86k, respectively. This is as expected, since cache-mapping policies are proprietary and $\mathcal{P}_{\text{h/m}}$ can only approximate them.

Conclusion. Based on these results, we conclude that CFAR's $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ projectors enable developers to accurately identify how the working set of third-party or their own code changes as a function of the workload, without requiring them to run elaborate benchmarks. Given that CFAR can extract distillates in < 30 mins, we envision such extraction and analysis of distillates to be a part of the regular development cycle (e.g., continuous integration), enabling developers to identify surprising performance behavior without having to write elaborate test suites.

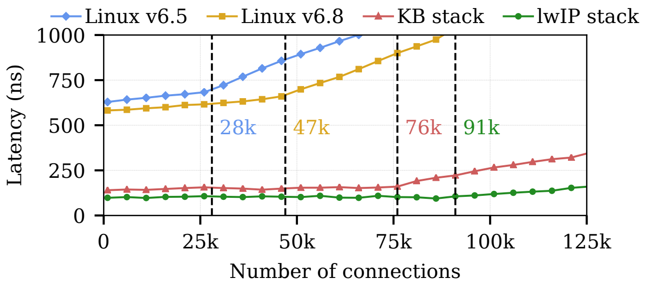

Figure 7: Measured latency for TCP packet processing as a function of the number of connections. 28k, 47k, 76k and 91k are the number of connections at which CFAR predicts persistent LLC misses for the Linux stack `v6.5`, Linux stack `v6.8`, the kernel-bypass stack and the lwIP stack, respectively.

5.2.2 How does data structure layout in code I call interact with my code?

We used the $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ projectors to analyze the cache usage of the hash table implementations from Vigor [72] and Klint [58]. We did not write this code, but we read it and thought we understood it fairly well. This is the kind of analysis that Alice would do to answer the other part of her question in §2: is the hash table the predominant cause of persistent LLC misses?

The projections proved our expectations about the performance of the two hash tables wrong: The two hash tables organize keys, values, and 4 metadata fields in slightly different ways: Vigor stores them as 6 distinct arrays, while Klint encapsulates all 6 fields into a single 64B `struct` and maintains a single array with elements of this `struct` type. At first glance, it appears—and did to us too—that the latter always leads to better locality and improved performance. However, it turned out that this is not always true.

Applying $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ on the `put`, `get`, and `delete` operations of the two implementations predicts the following: For a `put()` or `get()` call, both implementations bring 64B of data into the cache, but while Klint does so in one cache line, Vigor does so across 6 cache lines. When the table does not fit in the LLC, Klint suffers one LLC miss, while Vigor suffers 6. On the other hand, for a `delete()` call, both implementations touch 32B, but while Vigor brings only these 32B into the LLC, Klint brings in 64B. On closer inspection, this is because Klint must bring in at least one entire cache-line-aligned struct (it cannot bring in half a cache line); hence, it always brings in the value and 2 other metadata fields, even though it never accesses them. As a result, for a range of table occupancies, Klint overflows the LLC and suffers 1 miss, while Vigor fits in the LLC and suffers none. $\mathcal{P}_{\text{h/m}}$ predicts that this range begins at approximately 400k keys and ends at approximately 800k keys, at which point both implementations overflow the LLC.

To verify these predictions, we measured the latency and LLC misses incurred by the `put` and `delete` calls of the two implementations, for different table occupancies. Fig. 8 plots Vigor's latency overhead relative to Klint, as a function of table occupancy. As predicted: Klint `put` is consistently faster due to better locality. For occupancies of 400k-800k keys, Vigor `delete` incurs 30% lower latency than Klint; moreover, Vigor incurs no misses, while Klint incurs 1. There was one discrepancy between $\mathcal{P}_{\text{h/m}}$'s predictions and the outcome of our experiments: for occupancies above 860k, Vigor continued to incur 1 miss, whereas $\mathcal{P}_{\text{h/m}}$ predicted 3. We believe that this is due to Intel's stride prefetcher, which our current cache model does not consider.

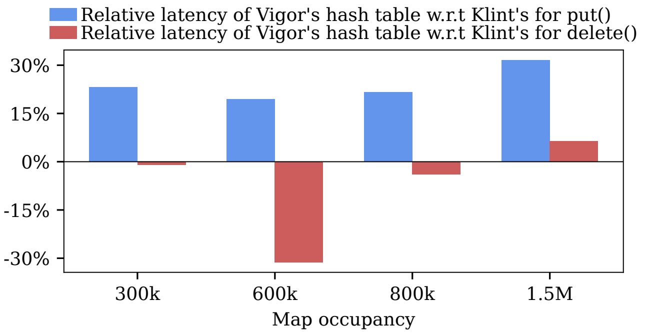

Figure 8: Relative latency (measured) of the Vigor hash table as compared to Klint's for `put()` and `delete()` calls. Positive numbers indicate that the Vigor table is slower and vice versa.

Conclusion. Data-structure developers often tailor their memory layout to different workloads [12–14]. While data-structure users are forced to resort to elaborate benchmarking to understand such subtle differences today (e.g., by measuring the performance of `put()` and `delete()` requests for all hash table sizes up to 1.5M) CFAR's $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ projectors can automatically make such subtle differences visible to data-structure users, allowing them to pick the implementation best suited for their expected workload.

5.2.3 Does my code lead to inefficient memory access patterns?

We now describe how CFAR's projections helped us uncover two inefficient cache access patterns in the kernel-bypass stack and the Hyperkernel's `mmap()` syscall, respectively.

Kernel-bypass stack: Motivated by the recent reorganization of the Linux TCP stack for cache efficiency—in particular, the `struct` that stores connection-specific data—we decided to see if the CFAR's projections could help us improve the performance of the kernel-bypass stack as well. To understand how the fast path of the kernel-bypass stack was accessing different fields of the connection-specific `struct` (named `struct pcb` in this stack), we wrote a simple projector that returned the offset (in cache lines) of each access within the `struct pcb` from the base address of the `struct`. The code below shows the 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)]
```

Applying this projector to the fast path's `rcv` and `snd` calls, made us realize that there was only a single access to the 5^{th} cache line in the `struct pcb`. The snippet below shows the list of cache-line accesses returned for the above calls:

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

Using the `filename:linenum` information that CFAR logs during symbolic execution (§4.1.1), we realized that the field being accessed was `keep_cnt_sent`, which was being updated on the `rcv` path to indicate that the connection was still live. We used this information to re-organize the `struct pcb`—we moved `keep_cnt_sent` into the first 4 cache lines and moved some of the timer fields (primarily used during retransmissions) to the 5^{th} line. The snippet below shows the list returned by the offset-computing projector after the change, which confirmed that the fast path only accessed the first 4 cache lines.

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

We evaluated the impact of this change by running the same experiment we ran in §5.2.1 where we measured the latency of the fast path as a function of the number of connections; Fig. 9 illustrates the results. We see that our fix has a significant impact on the fast path's connection scalability: touching one less cache line ensures that the stack can support 88k concurrent connections (as opposed to 72k) before suffering from a latency spike due to LLC misses.

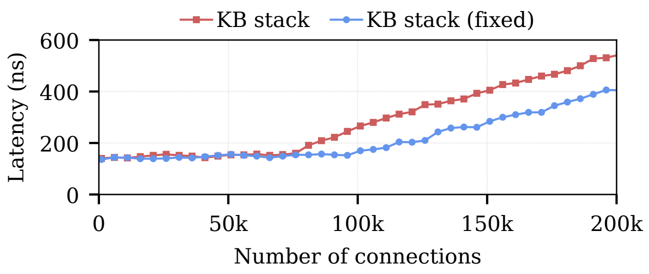

Figure 9: Measured latency as a function of the number of connections for the kernel-bypass stack, before and after our fix.

Hyperkernel `mmap()`: CFAR's $\mathcal{P}_{\text{scale}}$ projector enabled us to uncover and fix a subtle performance issue in Hyperkernel's `mmap()` implementation. The `mmap` code performs a four-level page walk, checking for permissions only before it allocates the final page. So, if it is called with invalid permissions, it does not exhibit incorrect behavior (it does not allocate the final page), but it still performs significant unnecessary work (allocates and zeroes out up to 3 new page-table pages, depending on where the walk fails). This brings up to $12KB$ of data into the L1 cache, and given that most servers today have an L1 cache of 32KB, this code unnecessarily pollutes up to roughly 40% of the cache.

Figs. 10 and 11 show parts of `mmap`'s projections before and after we fixed the above behavior. Consider Fig. 10: line 6 corresponds to the scenario where the walk fails at level 1 (no page-table page is allocated at that level for the target address), and the permissions are invalid; line 12 corresponds to the scenario where the code fails at level 2, and the permissions are valid. In the former case, the projection returns "201," while in the second, "202." So, the code touches almost the same number of cache lines, even though it should be accessing very different amounts of memory: in the former case it doesn't need to allocate any pages, whereas in the latter case it needs to start allocating at level 2. Now consider Fig. 11: line 4 corresponds to the scenario where the permissions are invalid; in this case, the projection always returns "3," i.e., the code touches only 3 cache lines.

```
1 def mmap_dcache_num_cache_lines(va,perm):
2     #State: pid, proc_tbl, pages
3
4     if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>39)&511)]:
5         if not (perm & PTE_PERM_MASK):
6             return 201
7         return 265
8
9     if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>30)&511)]:
10        if not (perm & PTE_PERM_MASK):
11            return 138
12        return 202
13    ....
```

Figure 10: Projection for buggy `mmap` code showing number of unique data cache lines accessed.

```
1 def mmap_optimized_dcache_num_cache_lines(va,perm):
2     #State: pid, proc_tbl, pages
3
4     if not (perm & PTE_PERM_MASK):
5         return 3
6     if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>39)&511)]:
7         return 265
8     if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>30)&511)]:
9         return 202
10    ...
```

Figure 11: `mmap` projection after fix.

Conclusion. Based on the above results, we conclude that the CFAR distillate, coupled with simple projectors enables developers to quickly inspect their code and identify hard-to-diagnose performance bugs without having to run elaborate benchmarks.

5.2.4 Can I prove/disprove the absence of leaks due to secret-dependent memory accesses?

Finally, we used CFAR's $\mathcal{P}_{\text{crypt}}$ projector to analyze the 8 OpenSSL algorithms listed in Table 3. The first 7 are the ones mentioned in the beginning of §5, while the last one is from a previous version of OpenSSL (`v1.1`). We included the latter because it is known to exhibit cache-based leakage (CVE-2018-0737 [54]), and we wanted to demonstrate CFAR's capability to identify this behavior (and none of the algorithms that we analyzed from the latest version of OpenSSL exhibit it).

Program	Remarks
OpenSSL 3.0 AES	Identified previously unknown branch-based leak
OpenSSL 3.0 ChaCha	Proved absence of secret-dependent branches, mem accesses
OpenSSL 3.0 ECDHE	Proved absence of secret-dependent branches, mem accesses
OpenSSL 3.0 MD5	Proved absence of secret-dependent branches, mem accesses
OpenSSL 3.0 MD4	Proved absence of secret-dependent branches, mem accesses
OpenSSL 3.0 Poly1305	Proved absence of secret-dependent branches, mem accesses
OpenSSL 3.0 SHA-256	Proved absence of secret-dependent branches, mem accesses
OpenSSL 1.1 RSA (CVE-2018-0737)	Reproduced known cache-based leak

Table 3: OpenSSL programs analyzed using CFAR's $\mathcal{P}_{\text{crypt}}$.

The $\mathcal{P}_{\text{crypt}}$ projector enabled us to confirm the cache-based leakage in OpenSSL `v1.1`, and also uncover a previously-unknown branch-based leakage in OpenSSL `v3.0.0`. In particular, the output projection revealed that the cipher-block unpadding function used by AES branches depending on secret input. To further investigate, we wrote another projector that shows the number of executed instructions; this revealed that the number of instructions executed by the function in question depends on the length of the input buffer's padding, making the code vulnerable to padding oracles. We reported this leakage to the maintainers who confirmed it [55], and we have submitted a pull request that has undergone multiple rounds of review and is in the final stages of getting merged.

Figs. 12, 13 show the output projection of the latter projector for the unpadding function before and after the fix. The former clearly indicates that the number of instructions depends on padding length, whereas the latter returns the same value independently from input.

```
def ossl_cipher_unpadblock_num_insns(buf, buf_len, block_size):

  if buf.padding_len == 0:
    return 44
  if buf.padding_len > block_size:
    return 48
  return 57 + 19*buf.padding_len
```

Figure 12: Projection showing how instruction count for AES's cipher unpadding function is a function of `buffer.padding_length`, which must remain secret.

```
def ossl_cipher_unpadblock_num_insns(buf, buf_len, block_size):
  return 2985
```

Figure 13: Projection showing instruction count for AES's cipher unpadding function after our fix.

Our experience with OpenSSL suggests that incorporating CFAR and its projectors into the development cycle could be useful to developers. For instance, we learnt that the specific branch leakage had been latent since OpenSSL `v1.1.1` (released in 2019) because the developer "just reused the code," and it had been missed despite the thorough code reviews that OpenSSL undergoes. Yet for the developer, a quick glance at the projection (before the fix) would have immediately revealed the problem. So, perhaps if distillates and projections were extracted regularly, e.g., as part of continuous integration, branch and cache leakage would be detected before making their way into production.

Conclusion. Since the distillate captures all information relevant to how a piece of code accesses memory, CFAR can help developers efficiently reason about more than just performance properties; $\mathcal{P}_{\text{crypt}}$ can help identify both branch- and cache-based leakage in cryptographic code (or prove their absence).

Evaluation summary. CFAR-extracted distillates are not only 100% accurate but also useful to system developers since they (and the resulting projections) provide developers with visibility into cache usage in a way that profilers and simulators cannot. In our evaluation, we demonstrated four concrete instances of such visibility and showed how CFAR enables developers to reason precisely about the cache usage of (1) their own, as well as (2) third-party code without having requiring them to run elaborate benchmarks, (3) quickly identify cache-inefficient access patterns that are otherwise hard to diagnose, and (4) inspect their code not only for performance bugs but also security vulnerabilities.

6 Related Work

Performance interfaces. CFAR is part of an ongoing effort towards augmenting systems code with *performance interfaces* [29–31, 46], i.e., interfaces that enable developers to efficiently reason about performance behavior, just like semantic interfaces (e.g., specifications, documentation) enable reasoning about functionality. While prior work focused on providing visibility into the latency behavior of third-party code, CFAR, by providing visibility into the usage of shared micro-architectural structures (namely the data and instruction caches), goes a step further, and enables developers to reason about the performance impact of incorporating third-party code into their systems.

Despite focusing on a different problem, CFAR leverages two key ideas from existing work on performance interfaces, particularly PIX [29]. First, just like PIX (and Freud [62]), CFAR also represents performance properties as human-readable, but also executable, programs. Second, CFAR's two-phased approach is similar to the separation between the PIX front- and back-end, which separates the performance properties of the code from the environment it runs in. This two-phased approach is beginning to see broader adoption; for instance, Performal [75] uses a similar approach to verify the performance of distributed systems, and LTC [46] uses this approach to provide visibility into the performance of hardware accelerators, which gives us confidence that our approach is the right one. That said, while the approach is similar, CFAR's key technical contributions, i.e., the abstractions of the distillates and projections, are specific to CFAR's problem statement of reasoning about cache usage.

Using automated program analysis to reason about performance properties of systems code. Given the recent advancements in automated program analysis techniques [26, 32, 36, 38, 63, 70] there have been several instances of recent work that use such analysis to analyze performance properties of systems code. Violet [27] uses it to find configuration bugs in large cloud applications, Bolt [30], and Castan [56] use it to analyze latency in software network functions, and finally Clara [60] proposes using it to analyze the performance impact of offloading programs onto smartNICs. However, in all of these cases, just using symbolic execution is not enough, one must develop the right analysis framework around it to reason about the specific property of interest. In CFAR, we use symbolic execution to extract information about the symbolic memory accesses, and then transform this information into program-like summaries and build projections on top of them, to enable developers to reason about the performance and security of their code.

Understanding cache usage of systems code. Given the ever growing gap between processor and memory speeds, understanding how systems code uses the cache has been extensively studied. However, we are not aware of any tool that (like CFAR) possesses predictive power across workloads. All prior tools we know of are limited to providing insights only about the workloads that the tool was run on.

We drew significant inspiration from work in the early 90s on abstract execution [39] and memory tracing [20]. Both these systems were designed to be able to replay the memory trace of a piece of systems code (just like CFAR's distillates), but only for concrete inputs. This is because their goal was to avoid having to store large memory traces required for computer architecture simulations, instead they sought to generate this trace on the fly. CFAR's distillate thus represents a generalized version of their work, that builds on advancements in automated program analysis.

More recent work has focussed on building better profilers [10, 16, 34, 42, 48, 57, 68] to help developers fix performance issues that they observe due to poor cache utilization. The key tradeoff that such systems explore is between ease of use, performance overhead and the level of detail at which they can analyze the execution of the given input workload. The most detailed memory profiler we know of is Memspy [48]. MemSpy uses a system simulator to execute an application which allows it to interpose on all memory accesses and build a complete map of the cache. Thus, MemSpy can account for and explain every single cache miss and using a processor accurate model can approximate memory access latencies. However, Memspy requires applications to be ported to its simulator which can be a painstaking task. Additionally its high performance overhead ensures that it can only be used to profile a limited number of input workloads. At the other end of the spectrum are profilers such as DMon [34]. These profilers work-off-the-shelf for almost all systems code, and have very low-enough overheads to run continuously in production. The downside is that they can only be used to monitor a very specific subset of events and cannot provide the visibility that MemSpy does. We see profilers as complementary to CFAR. CFAR's distillate and projectors allow developers to quickly understand which workloads might be of interest and cause unexpected cache behavior. Once they narrow this search space, they can use state-of-the-art profilers to study these workloads in great detail.

7 Conclusion

We presented CFAR, a technique and tool that enables system developers to precisely reason about how their code, as well as third-party code, uses the CPU cache. CFAR's approach consists of two phases: in the first phase, CFAR takes as input the code and extracts from it an abstract representation (a "distillate") that contains all the information on how the code accesses memory. In the second phase, CFAR uses simple programs ("projectors") that transforms the distillate into answers to specific questions about the code's cache usage. By introducing an abstraction (distillates) that precisely captures what a piece of code does to the micro-architecture as a function of its inputs; and a simple means—projectors—to query these abstractions, CFAR enables developers to efficiently answer diverse questions about the cache usage of systems code without having to delve into the implementation details of the code or run elaborate benchmarks.

We used CFAR to analyze a wide variety of systems code and demonstrate that it enables developers to not only identify performance bugs and security vulnerabilities in their own code, but also understand the performance impact of incorporating third-party code into their systems. For all the code we evaluated it on—which includes the fast path of the Linux kernel's TCP stack—CFAR's analysis completed in minutes, which makes us confident that extraction and analysis of distillates can be feasibly integrated into the regular software development cycle.

CFAR is publicly available as open-source software at https://dslab.epfl.ch/research/cfar.

8 Acknowledgements

We thank our shepherd Michael Stumm and the anonymous reviewers for their detailed feedback that significantly improved the paper. We are also grateful to the many people who provided helpful feedback on drafts of the paper at various stages: Ed Bugnion, Siddharth Gupta, Marios Kogias, James Larus, Solal Pirelli, and Sylvia Ratnasamy.

References

[1] Address Space Layout Randomization. https://en.wikipedia.org/wiki/Address_space_layout_randomization. [Last accessed on 2024-05-23].

[2] Ainsworth, S., and Jones, T. M. Software prefetching for indirect memory accesses. In *International Symposium on Code Generation and Optimization* (2017).

[3] Almeida, J. B., Barbosa, M., Barthe, G., Dupressoir, F., and Emmi, M. Verifying constant-time implementations. In *USENIX Security Symposium* (2016).

[4] AWS Elasticache. https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/AutoScaling.html. [Last accessed on 2024-05-23].

[5] Ayers, G., Nagendra, N. P., August, D. I., Cho, H. K., Kanev, S., Kozyrakis, C., Krishnamurthy, T., Litz, H., Moseley, T., and Ranganathan, P. AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers. In *Internation Symposium on Computer Architecture* (2019).

[6] Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Symposium on Operating Systems Design and Implementation* (2014).

[7] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. In *ACM SIGARCH Computer Architecture News* (2011).

[8] Boonstoppel, P., Cadar, C., and Engler, D. R. RWset: Attacking Path Explosion in Constraint-Based Test Generation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[9] Burger, D., and Austin, T. M. The simplescalar tool set, version 2.0. In *ACM SIGARCH Computer Architecture News* (1997).

[10] Cachegrind: A Cache and Branch-Prediction Profiler. https://valgrind.org/docs/manual/cg-manual.html. [Last accessed on 2024-05-23].

[11] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Symposium on Operating Systems Design and Implementation* (2008).

[12] Chilimbi, T. M., Davidson, B., and Larus, J. R. Cache-Conscious Structure Definition. In *International Conference on Programming Language Design and Implementation* (1999).

[13] Chilimbi, T. M., Hill, M. D., and Larus, J. R. Cache-Conscious Structure Layout. In *International Conference on Programming Language Design and Implementation* (1999).

[14] Chilimbi, T. M., and Larus, J. R. Using Generational Garbage Collection To Implement Cache-Conscious Data Placement. In *International Symposium on Memory Management* (1998).

[15] The Clang compiler. https://clang.llvm.org.

[16] Curtsinger, C., and Berger, E. D. Coz: Finding Code that Counts with Causal Profiling. *Commun. ACM* (2018).

[17] de Moura, L. M., and Bjørner, N. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[18] Dobrescu, M., Argyraki, K., and Ratnasamy, S. Toward Predictable Performance in Software Packet-Processing Platforms. In *Symposium on Networked Systems Design and Implementation* (2012).

[19] Dunkels, A. Design and Implementation of the lwIP TCP/IP Stack. Tech. Rep. 2:77, Swedish Institute of Computer Science, 2001.

[20] Eggers, S. J., Keppel, D. R., Koldinger, E. J., and Levy, H. M. Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. In *ACM SIGMETRICS Conference* (1990).

[21] Eyerman, S., Eeckhout, L., Karkhanis, T., and Smith, J. E. A Performance Counter Architecture for Computing Accurate CPI Components. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2006).

[22] Farshin, A., Roozbeh, A., Jr., G. Q. M., and Kostic, D. Make the Most out of Last Level Cache in Intel Processors. In *ACM European Conference on Computer Systems* (2019).

[23] Fuerst, A., Novakovic, S., Goiri, I., Chaudhry, G. I., Sharma, P., Arya, K., Broas, K., Bak, E., Iyigun, M., and Bianchini, R. Memory-Harvesting VMs in Cloud Platforms. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2022).

[24] Furia, C. A., Meyer, B., and Velder, S. Loop Invariants: Analysis, Classification, and Examples. *ACM Computing Survey* (2014).

[25] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed Automated Random Testing. In *International Conference on Programming Language Design and Implementation* (2005).

[26] Godefroid, P., and Luchaup, D. Automatic Partial Loop Summarization in Dynamic Test Generation. In *International Symposium on Software Testing and Analysis* (2011).

[27] Hu, Y., Huang, G., and Huang, P. Automated Reasoning and Detection of Specious Configuration in Large Systems with Symbolic Execution. In *Symposium on Operating Systems Design and Implementation* (2020).

[28] Hundt, R., Raman, E., Thuresson, M., and Vachharajani, N. MAO — An Extensible Micro-Architectural Optimizer. In *International Symposium on Code Generation and Optimization* (2011).

[29] Iyer, R., Argyraki, K., and Candea, G. Performance Interfaces for Network Functions. In *Symposium on Networked Systems Design and Implementation* (2022).

[30] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K., and Candea, G. Performance Contracts for Software Network Functions. In *Symposium on Networked Systems Design and Implementation* (2019).

[31] Iyer, R. R., Ma, J., Argyraki, K. J., Candea, G., and Ratnasamy, S. The Case for Performance Interfaces for Hardware Accelerators. In *Workshop on Hot Topics in Operating Systems* (2023).

[32] Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. TRACER: A Symbolic Execution Tool for Verification. In *International Conference on Computer Aided Verification* (2012).

[33] Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. E. TAS: TCP acceleration as an OS service. In *ACM European Conference on Computer Systems* (2019).

[34] Khan, T. A., Neal, I., Pokam, G., Mozafari, B., and Kasikci, B. DMon: Efficient Detection and Correction of Data Locality Problems Using Selective Profiling. In *Symposium on Operating Systems Design and Implementation* (2021).

[35] King, J. C. Symbolic Execution and Program Testing. *Journal of the ACM 19*, 7 (1976).

[36] Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., and Wintersteiger, C. M. Loop Summarization Using Abstract Transformers. In *Automated Technology for Verification and Analysis* (2008).

[37] Kumar, P., Dukkipati, N., Lewis, N., Cui, Y., Wang, Y., Li, C., Valancius, V., Adriaens, J., Gribble, S., Foster, N., and Vahdat, A. PicNIC: Predictable Virtualized NIC. In *ACM SIGCOMM Conference* (2019).

[38] Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. Efficient State Merging in Symbolic Execution. In *International Conference on Programming Language Design and Implementation* (2012).

[39] Larus, J. R. Abstract Execution: A Technique for Efficiently Tracing Programs. *Softw. Pract. Exp.* (1990).

[40] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Symposium on Networked Systems Design and Implementation* (2014).

[41] Analye and reorganize core networking structs to optimize cacheline consumption. Linux Kernel mailing list. `https://lore.kernel.org/netdev/20231129072756.3684495-1-lixiaoyan@google.com/`. [Last accessed on 2024-05-23].

[42] The Linux Perf Tool. `https://perf.wiki.kernel.org`. [Last accessed on 2024-05-23].

[43] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium* (2018).

[44] Lu, Q., Alias, C., Bondhugula, U., Henretty, T., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P., Chen, Y., Lin, H., and Ngai, T. Data Layout Transformation for Enhancing Data Locality on NUCA Chip Multiprocessors. In *International Conference on Parallel Architectures and Compilation Techniques* (2009).

[45] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. PIN: building customized program analysis tools with dynamic instrumentation. In *International Conference on Programming Language Design and Implementation* (2005).

[46] Ma, J., Iyer, R., Kashani, S., Emami, M., Bourgeat, T., and Candea, G. Performance interfaces for hardware accelerators. In *Symposium on Operating Systems Design and Implementation* (2024).

[47] Marinov, D., and Khurshid, S. TestEra: A Novel Framework for Automated Testing of Java Programs. In *International Conference on Automated Software Engineering* (2001).

[48] Martonosi, M., Gupta, A., and Anderson, T. E. MemSpy: Analyzing Memory System Bottlenecks in Programs. In *ACM SIGMETRICS Conference* (1992).

[49] Mowry, T. C., Lam, M. S., and Gupta, A. Design and Evaluation of a Compiler Algorithm for Prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (1992).

[50] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-Button Verification of an OS Kernel. In *Symposium on Operating Systems Principles* (2017).

[51] Github: State of the Octoverse 2022 - Programming Languages. https://octoverse.github.com/2022/top-programming-languages. [Last accessed on 2024-05-23].

[52] Olivo, O., Dillig, I., and Lin, C. Static Detection of Asymptotic Performance Bugs in Collection Traversals. In *International Conference on Programming Language Design and Implementation* (2015).

[53] OpenSSL. https://github.com/openssl/openssl. [Last accessed on 2024-05-23].

[54] OpenSSL CVE-2018-0737. https://github.com/advisories/GHSA-rj52-j648-hww8. [Last accessed on 2024-05-23].

[55] Pull request to fix constant-time violation in OpenSSL's Cipherblock Unpadding. https://github.com/openssl/openssl/pull/16323. [Last accessed on 2024-05-23].

[56] Pedrosa, L., Iyer, R., Zaostrovnykh, A., Fietz, J., and Argyraki, K. Automated Synthesis of Adversarial Workloads for Network Functions. In *ACM SIGCOMM Conference* (2018).

[57] Pesterev, A., Zeldovich, N., and Morris, R. T. Locating Cache Performance Bottlenecks Using Data Profiling. In *ACM European Conference on Computer Systems* (2010).

[58] Pirelli, S., Valentukonytė, A., Argyraki, K., and Candea, G. Automated Verification of Network Function Binaries. In *Symposium on Networked Systems Design and Implementation* (2022).

[59] Python3 Documentation: Lists. https://docs.python.org/3/tutorial/datastructures.html. [Last accessed on 2024-05-23].

[60] Qiu, Y., Kang, Q., Liu, M., and Chen, A. Clara: Performance clarity for smartnic offloading. In *ACM Workshop on Hot Topics in Networks* (2020).

[61] Ren, X. J., Rodrigues, K., Chen, L., Vega, C., Stumm, M., and Yuan, D. An Analysis of Performance Evolution of Linux's Core Operations. In *Symposium on Operating Systems Principles* (2019).

[62] Rogora, D., Carzaniga, A., Diwan, A., Hauswirth, M., and Soulé, R. Analyzing System Performance with Probabilistic Performance Annotations. In *ACM European Conference on Computer Systems* (2020).

[63] Saxena, P., Poosankam, P., McCamant, S., and Song, D. Loop-Extended Symbolic Execution on Binary Programs. In *International Symposium on Software Testing and Analysis* (2009).

[64] Sen, K., Marinov, D., and Agha, G. CUTE: a Concolic Unit Testing Engine for C. In *Symposium on the Foundations of Software Engineering* (2005).

[65] Soares, L., and Stumm, M. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Symposium on Operating Systems Design and Implementation* (2010).

[66] Sutherland, M., Gupta, S., Falsafi, B., Marathe, V., Pnevmatikatos, D., and Daglis, A. The NeBuLa RPC-Optimized Architecture. In *Internation Symposium on Computer Architecture* (2020).

[67] Tootoonchian, A., Panda, A., Lan, C., Walls, M., Argyraki, K. J., Ratnasamy, S., and Shenker, S. ResQ: Enabling SLOs in Network Function Virtualization. In *Symposium on Networked Systems Design and Implementation* (2018).

[68] Valgrind. https://valgrind.org. [Last accessed on 2024-05-23].

[69] Wei, H., Yu, J. X., Lu, C., and Lin, X. Speedup Graph Processing by Graph Ordering. In *ACM SIGMOD Conference* (2016).

[70] Xie, X., Chen, B., Liu, Y., Le, W., and Li, X. Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. In *Symposium on the Foundations of Software Engineering* (2016).

[71] Z3 Python API. https://github.com/Z3Prover/z3/blob/master/src/api/python/z3/z3.py. [Last accessed on 2024-05-23].

[72] Zaostrovnykh, A., Pirelli, S., Iyer, R. R., Rizzo, M., Pedrosa, L., Argyraki, K. J., and Candea, G. Verifying Software Network Functions with No Verification Expertise. In *Symposium on Operating Systems Principles* (2019).

[73] Zarandi, A. P., Sutherland, M., Daglis, A., and Falsafi, B. Cerebros: Evading the RPC Tax in Datacenters. In *International Symposium on Microarchitecture* (2021).

[74] Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O. S. N., Martinez, A., Liu, J., Simpson, A. K., Jayakar, S., Penna, P. H., Demoulin, M., Choudhury, P., and Badam, A. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Symposium on Operating Systems Principles* (2021).

[75] Zhang, T. N., Sharma, U., and Kapritsos, M. Performal: Formal Verification of Latency Properties for Distributed Systems. In *International Conference on Programming Language Design and Implementation* (2023).

[76] Zhang, Y., Ding, W., Kandemir, M. T., Liu, J., and Jang, O. A Data Layout Optimization Framework for NUCA-Based Multicores. In *International Symposium on Microarchitecture* (2011).

[77] Zhong, Y., Orlovich, M., Shen, X., and Ding, C. Array Regrouping and Structure Splitting Using Whole-Program Reference Affinity. In *International Conference on Programming Language Design and Implementation* (2004).

[78] Zhou, D., Yu, H., Kaminsky, M., and Andersen, D. G. Fast Software Cache Design for Network Appliances. In *USENIX Annual Technical Conference* (2020).