Automatically Reasoning About How Systems Code Uses the CPU Cache

Rishabh Iyer, Katerina Argyraki, and George Candea, *EPFL*

https://www.usenix.org/conference/osdi24/presentation/iyer

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Automatically Reasoning About How Systems Code Uses the CPU Cache

Rishabh Iyer, Katerina Argyraki, George Candea
EPFL, Switzerland

Abstract

We present a technique, called CFAR, that developers can use 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 implemented CFAR in an eponymous tool with which we analyze a performance-critical subset of four TCP stacks—two versions of the Linux stack, a stack used by the IX kernel-bypass OS, and the lwIP TCP stack for embedded systems—as well as 7 algorithm implementations from the OpenSSL cryptographic library, all 51 system calls of the Hyperkernel, and 2 hash-table implementations. We show 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 doing elaborate benchmarking.

CFAR is open-source and freely available at [58].

1 Introduction

System performance is important yet often poorly understood. Hence the recently proposed notion of a *performance interface* [30, 31, 47], defined by analogy to semantic interfaces (e.g., abstract classes, specifications, documentation) that 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. The goal of performance interfaces is to help developers efficiently reason about the performance behavior of both their own and 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 system kernels, device drivers, network stacks, hypervisors) 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, 50, 70] or whether the code's working set fits in cache [19, 23, 45, 67, 68, 77]. *Not* understanding this interaction can lead to performance bugs that are hard to diagnose, and can also result in unexpected performance behavior when using third-party code. For instance, a recent patch showed how the fast path of the Linux TCP stack had been experiencing a bloated cache footprint for over a decade, incurring slowdowns of up to 45% [42]; prior work has shown that applications may run up to 4× slower after calling into third-party code (e.g., a syscall) due to the callee's micro-architectural footprint [66, 74].

The goal of this work is to help system developers answer key questions about how their code and 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, 19, 23, 38, 45, 57, 67, 68, 70, 77, 78]. 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, 19, 23, 67, 68] and "Which workloads make the code's working set exceed the cache size?" [38, 57] without requiring developers to delve into the code's details or run elaborate, time-consuming benchmarks.

Answering the above questions requires visibility into how the code processes an *abstract* workload, so we look for abstractions that capture (in a succinct, precise, and human-readable manner) how the code interacts with the caches as a function of the workload. Our approach is in contrast to existing performance-analysis tools like profilers [10, 43, 59, 69] and cycle-accurate simulators [7, 9]: such tools can only provide insights into cache usage for the *concrete* workloads with which the code is profiled or simulated; they cannot provide visibility into how the code would behave for arbitrary, previously 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 [29], particularly for code that the developers did not write themselves.

We present Cache Footprint AnalyzeR (CFAR), a technique for processing a piece of systems code into answers to developers' questions about how that code uses the cache. This processing consists of two phases: In the first phase, CFAR takes as input the target code and extracts from it an abstract representation (a "distillate") of how the code accesses memory. In the second phase, CFAR uses simple programs ("projectors") to transform the distillate into answers to 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. The eponymous tool that implements the CFAR technique 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 level of visibility a developer seeks, enabling her to reason 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) $\mathcal{P}_{\text{scale}}$ computes how the amount of data the code brings into the cache (in bytes) varies as a function of the workload; (2) $\mathcal{P}_{\text{h/m}}$ computes, as a function of workload, whether memory accesses will hit or miss in the cache; and (3) $\mathcal{P}_{\text{crypt}}$ flags cryptographic code that branches on, or accesses memory in a way that depends on secret inputs, thereby flagging potential security vulnerabilities or proving their absence. $\mathcal{P}_{\text{crypt}}$ is an example that 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 standard performance properties. We envision developers contributing more such projectors to the CFAR tool, making it more useful over time. In stable state, developers will likely just use whatever ships with CFAR, extending it only when they cannot get the answers they seek.

We use CFAR to analyze a performance-critical subset of the transport layer of 4 TCP stacks—2 versions of Linux's stack (i.e., before and after the recent reorganization for cache efficiency [42]), a TCP stack used by the IX kernel-bypass OS [6], and the lwIP TCP stack for embedded systems [20]—as well as 2 hash-table implementations [60, 73], all 51 of the Hyperkernel's system calls [51], and 7 algorithm implementations from the OpenSSL cryptographic library [54]. We use the results to demonstrate how distillates and projectors enable developers to understand the cache usage of both their own and others' code, for unseen workloads, without running elaborate benchmark suites. As part of the evaluation, we also uncover a cache-inefficient data layout in the kernel-bypass TCP stack, an error path in the Hyperkernel `mmap()` system call (which, 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 means that extraction and analysis of distillates can be feasibly integrated into a real-world 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), then provide an overview of the CFAR approach (§3) and detail its design (§4). We then evaluate the CFAR prototype experimentally (§5), discuss related work (§6), and conclude (§7).

2 Motivation

In this section, we give an example of the kind of questions that system 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 an in-memory key-value store that has to be fast. 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 [20, 34, 75], 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 [41, 67, 79]; 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 [20, 67]. Finally, if both codebases were already highly optimized, she should avoid wasting time on code optimizations and replicate her key-value store across multiple machines [4].

2.2 Existing Tools Are Insufficient

Existing tools like profilers [10, 43, 59, 69] 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* 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 to the tools.

As a result, developers like Alice are forced to guess the answers to their questions based on (incomplete) information derived from profiling. Between cycle-accurate simulators and profilers, performance engineers typically prefer profilers because they are orders of magnitude faster, even if less precise. So, 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 in industry do to answer such questions: 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, 42].

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" [48, 53]: unexpected performance behavior often manifests only when input size (e.g., the number of concurrent connections) exceeds some threshold 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 metrics for performance coverage. While line coverage is used as a proxy for what fraction of the code's semantic behaviors is covered by tests, performance profiling does not have even 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 showed that the fast path of the Linux TCP stack had been accessing 50% more cache lines than it needed to, for over a decade, which was leading to performance degradation of up to 45% [42]. Similarly, initial work on predicting the working set of network functions ignored the impact of different packet sizes [19], and a study of Linux's system-call performance showed how a newly introduced configuration parameter can destroy spatial locality and lead to increased LLC misses [62]. In practice, developers like Alice often use incomplete performance profiling to guesstimate their system's cache usage, and then they overprovision resources for their system, to mitigate unexpected performance degradation [24]. This leads to lower system efficiency and inflated costs, and is not always effective.

Summary. Existing tools like profilers and cycle-accurate simulators are ill-suited to answering frequently-asked questions about cache usage, because they do not have *predictive power* across the space of possible 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 when applied to third-party code.

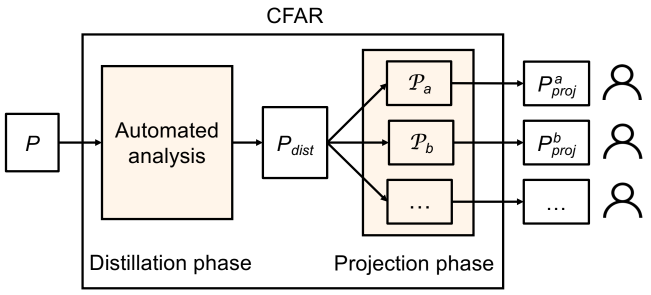

Figure 1: The CFAR workflow. P denotes a unit of systems code, P_{dist} denotes the corresponding distillate, $\mathcal{P}_i$ denotes the different projectors, and P^i_{proj} denotes the corresponding projections that provide answers to developers' questions about P's cache usage.

3 CFAR Overview

Answering questions about cache usage requires reasoning about the code. We therefore look for abstractions that precisely capture what the code does to memory (and thus the cache) as a function of its input (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 system call 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 much simpler 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., [22])—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 module P of a system's code and 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 vary as a function of the workload?" CFAR currently provides three such projectors, and we envision developers contributing more over time.

The two-phased workflow provides CFAR with the flexibility needed 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—

```c
 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 }
```

```python
 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 `sys_create` system call that creates a new file) and the corresponding data-accesses distillate.

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

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

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

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

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

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 standard 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, hypervisors), 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.

Notation	Description
P	A unit of code that can be invoked individually (e.g., system call, library function, standalone program). It takes as input I and has initial state S_0.
Ω	An ordered sequence of memory accesses (to symbolic addresses).
$\mathcal{P}_\pi$	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.

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 `%rsp`.

P's *distillate* P_{dist} is another (typically 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 with the example of the `sys_create` system call (Fig. 2, left) in the Hyperkernel [51]. First, each memory access in Ω is a tuple $<type, addr>$, where *type* can be read (`r`), write (`w`) or read-modify-write (`rmw`), and *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 an aligned offset 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 provides the exact 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, P_{dist} 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 $\pi(\Omega,\ldots)$ 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 $\pi(\Omega) = |\{\lambda(r) = \lfloor r/64 \rfloor : r \in \Omega\}|$, i.e., the number of unique 64-byte cache lines accessed by Ω.

We think of a function π as a question about cache usage (e.g., "How many memory accesses 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*: A function $\pi(\Omega, \ldots)$ defined by projector $\mathcal{P}_\pi$ is independent of the semantics of the code that produced Ω (or any of the other arguments). This code-agnostic nature makes projectors easy to express; for example, a developer can write the simple projector that defines $\pi(\Omega) = |\Omega|$ (mentioned above) to query the number of memory accesses performed by *any* P, without having to understand P's details.

The function π may take inputs beyond just Ω. For example, it may take as additional input a cache model, and compute the number of hits and misses incurred by Ω in the L1 data cache given that particular cache model. 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/or with different OS configurations (e.g., for different memory-page sizes that P may use).

3.3 Projections

A *projection* P^{π}_{proj} of P is another (typically simpler) program that takes the same input(s) I as P, 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}.

```
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.

Summary. CFAR provides abstractions to reason about what a program does to the memory hierarchy. If a program is a function P: $<I,S_o> \rightarrow$ semantic outputs, we say that the distillate is a function P_{dist}: $<I,S_o> \rightarrow \Omega$ that abstracts away everything that has to do with program semantics and preserves information about memory accesses. The projection is a function P^{π}_{proj}: $<I,S_o> \rightarrow \pi(\Omega)$ that computes an answer and abstracts away all unrelated information. This is a progression of abstraction steps, starting from the original program and arriving at the final projection. Each step takes the same arguments $<I,S_o>$, but returns a result that is increasingly more focused on the cache-usage question at hand.

4 CFAR Design

We now provide more details on the two phases of CFAR: distillation (§4.1) and projection (§4.2).

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 ① enumerates all feasible executions paths in P using automated program analysis; then ② obtains a binary execution trace for each such path; then ③ based on the results of these two steps, prepares an execution tree for the distillate; and lastly ④ optimizes this tree and produces P_{dist}.

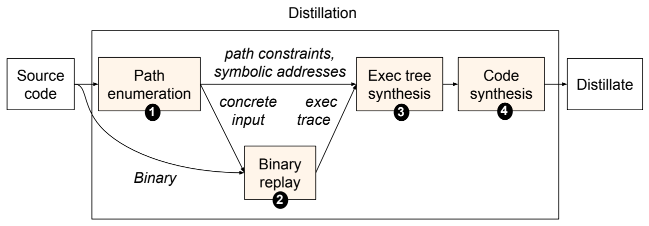

Figure 5: The four steps in CFAR's distillation process.

Our use of context-sensitive program analysis and binary replay ensures that CFAR can extract a precise distillate without requiring any effort on the part of the developer, but this also imposes limitations. Most notably, CFAR is subject to the scalability limitations of such program analysis and is thus not ideal for reasoning about multi-threaded code or about code with loops whose bounds 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 we note here that, despite these limitations, CFAR is able to provide valuable information about cache usage that is otherwise unavailable, and do so for a wide variety of systems code (§5).

4.1.1 Step ①: Path enumeration

To enumerate all the paths in P, CFAR uses *exhaustive* symbolic execution [11, 26, 36, 65]. This is a context-sensitive 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 predicates of each conditional along α; (2) a concrete input I_α that exercises this path, obtained by asking an SMT constraint solver [18] for a satisfying assignment to the free 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 (4) a corresponding `file: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 symbolic-execution engines, analyzes the code at the IR level, which in KLEE's case is LLVM [16]. CFAR can therefore handle any code that is compiled to LLVM and can be handled by KLEE. Our KLEE modifications and additions total ~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. 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 pops to/from the stack are not present in an SSA IR, and thus not captured when analyzing P in its 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 a corresponding execution trace X_α for each path α in P. For each machine instruction executed in X_α, CFAR saves the program counter, the instruction opcode (e.g., `mov`, `push`, `pop`), the concrete memory addresses accessed, and the corresponding `file:linenum` debug information inserted into the binary by the instrumentation.

We deliberately split the CFAR analysis into a source-based and a binary-based step. On the one hand, it is easier to extract symbolic expressions for memory operations by analyzing the source code 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 executing the binary symbolically (e.g., with S2E [15]). To answer with certainty whether this is possible, 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 [46] to instrument binaries. Our Pintool consists of ~350 lines of C++.

4.1.3 Step ③: Execution tree synthesis for P_{dist}

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, resulting in 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., from its entry point, if P is a function or a system call) to 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 root to leaf in the tree, and vice-versa. Each internal node of 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: Synthesis of the P_{dist} distillate

The final step in CFAR's distillation process consists of summarizing loop-related memory access patterns, along with other improvements for human readability of P_{dist}. Symbolic execution, by default, unrolls loops and thus produces a separate execution path for each loop iteration. This leads to bloated distillates that are hard to read and contain redundant information, particularly if the code's memory-access pattern does not change meaningfully across loop iterations.

Summarizing a loop entails representing the effects of that loop without representing all its iterations explicitly. Conceptually, the goal of this step is to eliminate from the execution tree the subtrees induced by loop unrolling. This step does not elide or lose any information contained in the distillate—it only optimizes the distillate's control flow for human readability. By definition, any projection derived from P_{dist} will retain P's control flow, as reflected in the distillate.

While automatically summarizing loops in general is undecidable [25], studies have shown that there exist four common categories of loops that relate to data locality issues in systems code [35]. Therefore, CFAR contains loop-summary templates for these four categories of loops—two that traverse array-like data structures, and two that traverse pointerchasing data structures (e.g., linked lists, trees). All four categories require the loop body to not branch on the specific value of the iteration counter, and require the loop to have a maximum of two termination predicates, one in the loop definition and at most one `break` in the body. Each loop template has a corresponding loop summary.

For loops that do not match a template, the distillate presents them in unrolled form—still correct, just less human-readable. We thus call this "best-effort" loop summarization.

After loop summarization and a few other optimizations for readability, CFAR transforms the tree into a program that represents P_{dist}. This program takes the same input as P. Every internal tree node generates 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, the returned symbolic memory-access trace is either Ω_α^{instr} or Ω_α^{data}.

Fig. 6 shows a snippet for the P_{dist}^{data} distillate of the `memcmp` function in the C standard library. Our CFAR prototype uses Python to represent distillates, because it is one of the most widely used languages [52] and has an easy-to-understand syntax. The example illustrates CFAR's loop summarization for a loop that belongs to the first category of loops mentioned above. CFAR uses first-order logic to summarize loops with primitives from Z3's Python API [72]. The predicate that starts on line 3 identifies the smallest index `i` (bounded by `len`) at which the two strings differ. The distillate shows that the memory accessed by `memcmp` 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 contains *all* the information on P's memory-access behavior. The answer to a developers' cache-usage question can therefore be found in the distillate, but it is buried among details that may not be relevant to that question. The projection phase turns distillates into focused, actionable answers that are not clouded by details irrelevant to the question being asked.

4.2.1 General overview

Developers write projectors in the form of programs that take as input a tuple $<\Omega, C>$, along with possibly other projector-specific parameters, such as the cache model mentioned in §3.2. Ω is a symbolic memory-access trace, and C is a constraint on the variables that appear in the symbolic addresses of Ω, in the form of a first-order logic expression.

We expect most projectors to ignore C and implement a function of just Ω, like $\pi(\Omega) = |\{\lambda(r) = \lfloor r/64 \rfloor : r \in \Omega\}|$. We therefore did not mention the C parameter in §3.2, for clarity of presentation, but, to answer all cache-usage questions, the more general function $\pi(\Omega, C)$ is sometimes needed. For example, determining if all memory accesses are cache-line-aligned requires, in the general case, both Ω and C. The origin of this constraint C is the path constraint that causes the original program to execute the memory accesses in Ω. The branch predicates in distillates are such constraints. The constraint can be simple, like `0≤idx<128`, or more sophisticated, stating for example that the value stored at a particular memory location is non-zero: `[file_tbl+40*fn+8]`$\neq 0$. In our prototype, Ω and C are Python lists of Z3 expressions [72].

To produce a projection, CFAR takes a projector program $\mathcal{P}_\pi$ and a distillate P_{dist}, and synthesizes a new program P_{proj}^{π}. For each branch in P_{dist}, the P_{proj}^{π} program has the same branch as P_{dist} but, instead of returning Ω (as the distillate does), it returns the value of invoking $\mathcal{P}_\pi(\Omega, C, \ldots)$. In other words, the projection P_{proj}^{π} has the same control flow as the distillate but, instead of calculating a memory-access trace, it calculates a specific cache-usage property of that trace.

4.2.2 Example projectors: $\mathcal{P}_{\text{scale}}$, $\mathcal{P}_{\text{h/m}}$, and $\mathcal{P}_{\text{crypt}}$

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

While the questions answered by these projectors are non-trivial, the projectors themselves are straightforward to write. For example, we express the functionality of both $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{crypt}}$ in less than 100 lines of Python. While $\mathcal{P}_{\text{h/m}}$ requires ~800 lines, almost 600 of those are a Python translation of the cache model from the `gem5` cycle-accurate simulator [7]. In §5.2, we show how a simple 5-line projector helped us identify a performance bug in a TCP stack used by IX [6].

$\mathcal{P}_{\text{scale}}$ computes cache footprint based on the symbolic Ω for a given input class. $\mathcal{P}_{\text{scale}}$ first determines which addresses in Ω change if the value of the input changes. It then uses an SMT solver to check the alignment of these addresses and determine the number of unique bytes touched by the accesses to those addresses, and produces the result as a human-readable formula. For instance, applying $\mathcal{P}_{\text{scale}}$ to the `sys_create` distillate in Fig. 2, for the input class that corresponds to successful creation (line 20), yields the formula `8*fd + 32*fn`. This formula says that, in a sequence of successful `sys_create` calls, the cache footprint will increase by 8 bytes for each distinct `fd` argument value and by 32 bytes per distinct `fn` argument value. This is how `fd` and `fn` influence the cache footprint, and this is exactly the information that Alice wanted to know for keys and network connections in §2. $\mathcal{P}_{\text{scale}}$ can be used, for instance, to quickly determine when the code's working set will overflow the cache.

$\mathcal{P}_{\text{h/m}}$ is more sophisticated and takes four projector-specific parameters: a workload size W, an input set cardinality N, a probability mass function PMF, and a cache model. Its goal is to compute the number of hits and misses experienced by a workload of W inputs that can take on any of N distinct types, distributed within the workload according to the PMF, when using a cache that works according to the model. In the scenario we will explore in §5 for a TCP network stack, W would be the number of packets in a trace, and N the number of unique connections—what distinguishes packet types is which connection they belong to. The PMF would be the relative distribution of packets among the N connections.

$\mathcal{P}_{\text{h/m}}$ first produces a workload of W symbolic inputs satisfying the PMF. For example, if W=5 and N=2 and the PMF is $<0.4, 0.6>$, the workload would be $<\lambda_1, \lambda_2, \lambda_1, \lambda_2, \lambda_2>$ or some other variant that satisfies the PMF. Then, for each symbolic input in the workload, $\mathcal{P}_{\text{h/m}}$ iterates through Ω and sends each memory access to the cache model; the access may be a function of the λ_i symbolic input, or independent of it. $\mathcal{P}_{\text{h/m}}$ records, for each access, whether it is a hit or a miss.

Since there are multiple workload variants that satisfy the PMF, the process above repeats, with alternate variants, until the resulting hits/misses counts are statistically significant.

$\mathcal{P}_{\text{h/m}}$ can be thought of as a symbolic, trace-based cache simulator. A major challenge is, when dealing with symbolic addresses, $\mathcal{P}_{\text{h/m}}$ cannot compute set-associativity conflicts precisely. Instead, $\mathcal{P}_{\text{h/m}}$ *approximates* them by allocating an unconstrained, symbolic memory address (typically the base of a data structure) randomly to a set in the cache, and then mapping all relative addresses as offsets from that base. For example, if the address γ is randomly mapped to set μ, then the address $\gamma + 64$ would be deterministically mapped to set $(\mu + 1)$ `mod` # of sets in the cache. In §5 we will see that this approximation works well when compared to real hardware.

By default, $\mathcal{P}_{\text{h/m}}$ provides a 3-level inclusive cache with a next-line prefetcher whose size and set associativity at each level is configurable. The default PMF is uniform, and W=100N. $\mathcal{P}_{\text{h/m}}$ assumes that the memory trace Ω does not update initial program state S_0 that influences addresses in Ω.

$\mathcal{P}_{\text{crypt}}$ is an example of a projector that also takes the path constraint C into account. Its goal is to answer the question of whether there are any data accesses to secret-dependent memory addresses or secret-dependent branches, both of which are known sources of side channels [3]. $\mathcal{P}_{\text{crypt}}$ takes, as a projector-specific input, a list of program inputs that constitute secrets. It then uses an SMT solver to determine which (if any) of the memory addresses in Ω are influenced by secrets. It then checks whether any of the secrets appear in the path constraint C. If it finds any, then $\mathcal{P}_{\text{crypt}}$ returns `file:linenum` debug information for the corresponding branch or memory access, as well as the path constraint that leads to it. If none found, then $\mathcal{P}_{\text{crypt}}$ states that the code does not have secret-dependent branch instructions or data accesses.

$\mathcal{P}_{\text{crypt}}$ cannot check for all types of cache-based leakage. For example, leakages due to speculatively executed instructions [44] 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 CFAR from 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 [39], loop-extended symbolic execution [64], loop summaries [27, 71], loop invariants [33], and symbolic abstract transformers [37].

Any CFAR prototype will ultimately inherit the power of its underlying SE engine. Since our prototype relies on KLEE [11], 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 systems, like the Memcached or Redis key-value stores. Nevertheless, we show in §5 that CFAR extracts useful distillates for key components of complex systems code (e.g., for data structures whose cache footprint is a common source of performance problems).

Using CFAR for code that is not amenable to exhaustive symbolic execution: CFAR's reliance on exhaustive symbolic execution means that automatically extracting complete distillates is not always feasible. We now discuss how developers can obtain useful results with CFAR even in such cases.

A simple approach is to *constrain the input space*, e.g., constrain CFAR to inputs that trigger the "fast path" through the code, since that is a common target of performance analysis. For instance, if the code of interest is an IP-packet 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 obtain 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 typically does not carry IP options). Focusing on the cache usage of the fast path is common practice today; for instance, the recent reorganization of the Linux TCP stack was based entirely on the requirements of the TCP fast path [42].

For constraining the input space, CFAR provides an interface similar to KLEE's [11], with which developers can provide constraints on arbitrary program variables. In our evaluation, we use this approach to analyze code that is not amenable to exhaustive symbolic execution (e.g., the Linux TCP stack), and the results are compelling despite the constrained input space. Constraining the input space requires the developer to have some knowledge of, for instance, what typical/fast-path inputs look like. However, it does not require knowledge of the code's internal details, because these are explored automatically by CFAR.

An alternate approach is to run CFAR with a specified time budget. When the time limit is reached, CFAR outputs a partial distillate that returns the exact sequence of symbolic memory accesses performed by the code along the explored execution paths. While this approach does not require developer knowledge, the downside is that CFAR may not explore all meaningful execution paths in the given time budget. The CFAR prototype offers this time-budget feature, but we did not use it in any of the experiments in our evaluation.

Prototype cannot aggregate across input classes: In the current projection model (§4.2), a projector instance gets to see only the Ω corresponding to one input class. Our CFAR prototype does not yet support sharing state across different instances of projectors, and thus does not support aggregating measures across multiple input classes. This support is simple to add, we just have not encountered the need for it yet.

Cannot account for inter-process interactions: Projectors cannot answer questions that span multiple processes. Also, we assume that P is small enough to not have its execution interrupted by preemption. The distillate P_{dist} is always correct with respect to P, but the predictions made based on this distillate alone will miss cache accesses performed by code other than P during a preemption. In other words, if the period of interest includes a preemption, when a projector looks at Ω, it does not get the full picture, because P is not the only code that interacts with the micro-architecture.

Limitations due to proprietary hardware details: CFAR employs binary instrumentation to obtain an execution trace. Such instrumentation can only reveal instructions that the processor retires (i.e., finishes executing); it does not reveal instructions that were executed as a result of incorrect speculation, such as a mispredicted branch. Speculated instructions nevertheless could impact the cache, even if their semantic effects are undone. Since CFAR does not see those accesses, 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 the out-of-order pipelines of commercial processors are proprietary.

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).

Evaluation targets. We used CFAR to analyze the fast path of the transport layer of 4 TCP stacks: 2 versions of Linux's stack (before and after the recent reorganization for cache efficiency [42]), a TCP stack used by the IX kernel-bypass OS [6], and the lwIP TCP stack for embedded systems [20]. We also analyzed 2 hash-table implementations [60, 73], all 51 system calls in the Hyperkernel [51], and 7 algorithm implementations in OpenSSL 3.0.0 [54]. For the Linux TCP stack, we analyzed the stable versions before and after the reorganization (`v6.5` and `v6.8`). For all other code, we analyzed the latest stable version. IX uses the lwIP stack as a starting point, but heavily modifies the internal data structures and timer management [6].

We demonstrate that CFAR provides actionable cache-usage information for a broad spectrum of systems code. 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, since they are both amenable to manual (but not automated) program analysis. At the other end of the spectrum are the four transport-layer implementations, which were not written to be amenable to any form of program analysis.

Setup. All experiments ran on an Intel Xeon E5-2690 v2 CPU at 3.30GHz with 25.6MB of LLC and 252GB of DRAM, with Ubuntu 22.04 and Linux kernel v5.4. The CFAR prototype incorporates a modified version of KLEE 2.1.

5.1 Does CFAR Work?

There are two key aspects to this question: does CFAR 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 half the execution paths of each target, constructed inputs that exercised each path, counted the number of instructions and memory accesses executed while running with each concrete input, and then compared this number to the one predicted by the target's distillates.

The error was always **zero**, across all programs and inputs. That is, the number of instructions counted during real execution always equaled the number of memory accesses predicted by the instruction-accesses distillate for the given input, and the number of memory accesses counted during real execution always equaled the number of memory accesses predicted by the data-accesses distillate for that input. Additionally, CFAR's distillates correctly predicted every instruction-memory and data-memory address accessed by the code.

5.1.2 Time to extract distillates

Table 2 shows how long CFAR takes to extract distillates: for all programs, the analysis completes in less than 30 min. The longest times are for the Vigor hash table and the `echde` key-generation algorithm in OpenSSL; they are the only ones that take more than 15 min. This is because, for these programs, symbolic execution needs to unroll long loops that iterate over the hash map and that compute co-prime numbers, respectively. For all programs, the binary replay, execution-tree synthesis, and code synthesis take approximately 2-3 min in total. The dominant component of CFAR's analysis time—and the one that varies across programs—is symbolic execution.

Program	P_{dist} **extraction time**
Linux TCP ingress	11 min
Linux TCP egress	14 min
IX TCP ingress	5 min
IX TCP egress	7 min
lwIP TCP ingress	4 min
lwIP TCP egress	5 min
Hyperkernel syscalls (51 total)	Avg: 4 min / Max: 7 min
OpenSSL primitives (7 total)	Avg: 9 min / Max: 22 min
Vigor hash table	28 min
Klint hash table	12 min

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 cases of CFAR answering important questions that developer Alice cannot readily answer with the state of the art: How does my code's working set vary with the workload (§5.2.1)? I want to use a third-party data-structure library, but how does it interact with my cache (§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 vary 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 four TCP stacks. Recall that this is the question that Alice wanted to answer in §2, but could not.

We constrain the input space as discussed in §4.3: focus the analysis solely on 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 pick this particular class of packets because it represents a large fraction of packets processed by the TCP stack, on the path for which performance matters the most. The recent re-organization of the Linux TCP stack was focused entirely on this fast path [42].

First, we used $\mathcal{P}_{\text{scale}}$ to figure out the number of unique cache lines touched by the TCP fast path, for symbolic packet contents. The answer was 4, 5, 8, and 12 unique cache lines for the lwIP, kernel-bypass, Linux stack `v6.8` and `v6.5`, respectively. $\mathcal{P}_{\text{scale}}$ provides this information automatically, whereas benchmarking or code inspection would have a hard time producing it, because it cannot 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 used it to predict when incoming packets were likely to suffer consistent cache misses due to the working set overflowing the LLC. 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 `v6.5`, respectively. The small differences between these predictions and simple capacity-based calculations (e.g., 25.6M LLC / (64*4) = 100K connections for lwIP) are due to $\mathcal{P}_{\text{h/m}}$ being able to account for 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. 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 packet-processing latency as a function of the number of connections. For each of the four 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 higher latency, the latency for the lwIP stack increases by only 13ns from 1K to 86K connections, and it increases by 50ns from 86K to 125K connections. 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 difference is expected, because of $\mathcal{P}_{\text{h/m}}$'s set-associativity conflict approximation (§4.2.2) and because cache-mapping policies are proprietary, so $\mathcal{P}_{\text{h/m}}$'s cache model is imprecise.

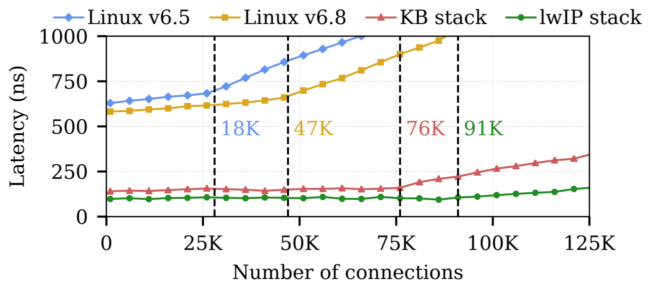

Figure 7: Measured latency for TCP packet processing as a function of the number of connections. CFAR predicted consistent LLC misses to start occurring at 28K, 47K, 76K and 91K connections for the Linux TCP stack `v6.5`, Linux TCP stack `v6.8`, the kernel-bypass (KB) stack and the lwIP stack, respectively.

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 code 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 under 30 minutes, such extraction and analysis of distillates can become part of the regular development cycle (e.g., be made part of a continuous-integration pipeline), enabling developers to identify surprising performance behavior early and with relatively little effort.

5.2.2 How does third-party data-structure library code interact with my cache?

System developers often want to use third-party data-structure implementations, but are anxious about how that code will interfere with their own usage of the cache. We used the $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ projectors to show how one can get cache-usage information for the hash-table implementations from Vigor [73] and Klint [60]. The analysis we show here can drive the choice between using one library vs. the other. We did not write the two libraries, but we read their code and *thought* we understood it fairly well.

For the CFAR analysis, we constrained the input space by fixing the maximum capacity of the hash tables to 64K entries. The resulting distillates had a different branch for each possible number of hash collisions, which is bound by the maximum capacity (thus, 64K cases). The conclusion of the CFAR analysis is therefore formally correct only for this maximum capacity. However, both hash-table implementations take the capacity as a configurable parameter, and the resulting memory-access pattern is independent of table capacity. We validated this through code inspection, we just cannot prove it formally using symbolic execution. Thus, like a developer, we proceeded to use $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ assuming that the distillates for the two hash tables provide valid predictions for any capacity, as long as the number of collisions does not exceed 64K. (The experiments validated this assumption.)

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 packs all 6 fields into a single 64B `struct` and maintains a single array with elements of this `struct` type. At first glance, it appears that the latter always leads to better locality and thus improved performance. However, it turned out that this is not always true.

Applying $\mathcal{P}_{\text{scale}}$ and $\mathcal{P}_{\text{h/m}}$ to the `put()`, `get()`, and `delete()` operations of the two implementations predicts the following: For a `put()` or `get()`, both implementations bring 64B of data into the cache, but Klint does so in 1 cache line, while Vigor does so across 6 cache lines. When the table does not fit in the LLC, Klint suffers 1 LLC miss, while Vigor suffers 6. On the other hand, for a `delete()` call, both implementations touch the same 32B. However, Klint packs them together with other fields into a cache-line-aligned 64B `struct`, so it must bring the full 64B into the LLC, then update the 32B for invalidating the entry; the remaining 32B belonging to the deleted entry will never be reused. For Vigor, even though it brings a full 64B into the LLC, the other 32B belong to a still-valid entry and are likely to be reused, and thus the cost is amortized. 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. We configured a capacity of 2M entries for both hash tables. 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. Yet, for occupancies of 400K-800K keys, Klint `delete()` has 30% worse latency than Vigor. As predicted, Vigor incurs no misses in this range, while Klint incurs 1 per `delete()` call. There was one discrepancy between $\mathcal{P}_{\text{h/m}}$ predictions and the outcome of our experiments: for occupancies above 860K, $\mathcal{P}_{\text{h/m}}$ predicted 3 misses per `delete()` for Vigor, whereas we measured only 1 per call. We believe this to be due to Intel's stride prefetcher, which our current cache model does not take into account.

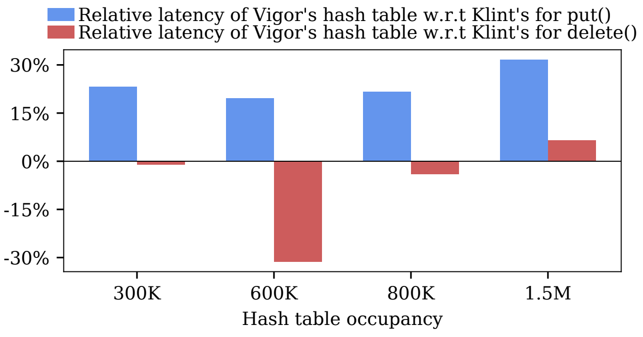

Figure 8: Relative latency (measured) of the Vigor hash table as compared to Klint's, for `put()` and `delete()` calls. Positive numbers indicate that the Vigor table is slower, and vice-versa.

Conclusion. Data-structure libraries often tailor their memory layout to different workloads [12–14]. Those who use the data structures need to understand these choices and the differences between different implementations. Using benchmarks can be tedious (e.g., in the present example, measuring the performance of `put()` and `delete()` for hash-table sizes up to 2M). Instead, CFAR projectors can quickly reveal the differences between how the implementations affect the cache across the entire range of inputs, allowing developers to pick the implementation best suited for their expected workload.

5.2.3 Does my code lead to inefficient access patterns?

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

Kernel-bypass (KB) TCP stack: Motivated by the recent re-organization of the Linux TCP stack for cache efficiency—in particular, the `struct` that stores connection-specific data—we decided to check if 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` (Fig. 9).

```
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)]
```

Figure 9: Projector to compute the offset within the pcb structure.

Applying this projector to the fast path's `rcv()` and `snd()` calls revealed that there was only a single access to the 5th cache line in the `struct pcb`. Fig. 10 shows the list of cache-line accesses returned for `rcv()` and `snd()`, respectively.

```
# 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]
```

Figure 10: Cache-line accesses for `rcv()` and `snd()` on fast path.

Using the `file: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. To optimize this, we re-organized the `struct pcb` by moving `keep_cnt_sent` into the first 4 cache lines and moved some of the timer fields (primarily used during retransmissions) to the 5th line. Fig. 11 shows the list returned by the pcb-offset projector after this 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]
```

Figure 11: Cache-line accesses after our pcb optimization.

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. 12 shows the results. Our optimization has a significant impact on the fast path's connection scalability: touching one less cache line enables the TCP stack to support 88K concurrent connections (instead of only 72K) before suffering from a latency increase due to LLC misses.

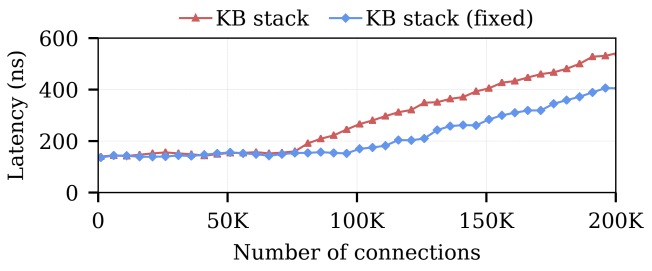

Figure 12: Before-and-after optimization: Latency as a function of the number of connections, for the kernel-bypass (KB) TCP stack.

Hyperkernel `mmap()`: CFAR 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 performs significant unnecessary work (allocates and zeroes out up to 3 new page-table pages, depending on where the walk stops), even if it does not exhibit incorrect behavior (i.e., does not allocate the final page). This brings up to 12KB of data into the L1 cache, which is more than 37% of the 32KB L1 cache in a modern server, so doing this unnecessarily pollutes the cache.

Fig. 13 shows part of the projection for `mmap()` resulting from a $\pi(\Omega) = |\{\lambda(r) = \lfloor r/64 \rfloor : r \in \Omega\}|$ projector. Line 6 corresponds to the scenario where the permissions are invalid, and the walk fails at level 1 (i.e., no page-table page is allocated at that level for the target address). Line 12 corresponds to the scenario where the permissions are valid, and the walk fails at level 2. In the former case, the code touches `201` cache lines, whereas in the latter it touches `202`. So, even though the code need not allocate any pages in the former case, it touches almost identical numbers of cache lines in both cases.

```
 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 13: Unique data cache lines accessed by `mmap()`.

We fixed the code and ran CFAR again. Fig. 14 shows the new projection: in the invalid-permissions scenario (line 4), the code now touches 3 (instead of >200) cache lines.

Conclusion. The results show that the CFAR distillate, coupled with simple projectors, enables developers to efficiently (i.e., without benchmarking) inspect systems code and identify performance bugs that are otherwise hard to diagnose.

```
def mmap_optimized_dcache_num_cache_lines(va,perm):
    #State: pid, proc_tbl, pages

    if not (perm & PTE_PERM_MASK):
        return 3
    if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>39)&511)]:
        return 265
    if [pages + [proc_tbl+320*pid+16]*4096 + 8*((va>>30)&511)]:
        return 202
    ...
```

Figure 14: `mmap` projection after fix.

5.2.4 Can I prove/disprove the absence of side channels caused by 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 [55]), and we wanted to test CFAR's ability to identify this behavior (none of the algorithms we analyzed in the latest version of OpenSSL exhibit it). The $\mathcal{P}_{\text{crypt}}$ projector indeed confirmed the cache-based leakage in OpenSSL `v1.1`.

Program	Results
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	Reproduced *known cache-based leak* (CVE-2018-0737)

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

We also uncovered a previously-unknown branch-based side channel in OpenSSL `v3.0.0`. The $\mathcal{P}_{\text{crypt}}$ projection revealed that the cipher-block unpadding function used by AES had secret input in the path constraint. To further investigate, we wrote another projector that counts the number of executed instructions. This revealed (Fig. 15) 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.

```
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 15: Instruction-count projection reveals that the number of instructions executed by AES's cipher unpadding is influenced by `buffer.padding_length`, which is a secret input.

We reported this to the maintainers, who confirmed it [56]. We submitted a fix, which has undergone multiple rounds of review and is now in the final stages of getting merged.

The instruction-count projection after the fix shows that the number of instructions is now independent of input (Fig. 15), thus proving that it achieves constant-time execution [3].

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

Figure 16: AES instruction count after our fix.

Our experience with OpenSSL suggests that incorporating CFAR and its projectors into the development cycle would be beneficial. As it turns out, the side channel we found had been latent in OpenSSL since `v1.1.1`, which was released in 2019. It persisted despite the thorough code reviews that OpenSSL undergoes. Yet, a quick glance at the projection before the fix would have immediately revealed the problem. Perhaps, if distillates and projections were extracted regularly, e.g., as part of continuous integration, more side channels could 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. $\mathcal{P}_{\text{crypt}}$ helps identify both branch- and cache-based leakage in cryptographic code (or prove their absence).

Evaluation summary. CFAR-extracted distillates are 100% accurate. They are useful to system developers because, together with projectors, they give visibility into cache usage in a way that profilers and simulators cannot. Our evaluation shows four concrete instances of such visibility and shows how CFAR enables developers to (1) reason precisely about the cache usage of code they or others wrote, without having to run elaborate benchmarks; (2) quickly identify cache-inefficient access patterns that are otherwise hard to diagnose; and (3) analyze code not only for performance bugs but also for cache-based security vulnerabilities.

6 Related Work

Performance interfaces. CFAR is part of an ongoing effort to augment systems with *performance interfaces* [30–32, 47]; these are meant to enable developers to efficiently reason about performance, just like semantic interfaces (abstract classes, specifications, documentation) enable reasoning about functionality. Some of this work [30, 31] provides visibility into the latency of software network functions, assuming a simple cache model that is appropriate for that particular domain. By providing visibility into the usage of shared micro-architectural structures (namely, the data and instruction caches), CFAR goes a step further: it enables developers to reason about the performance of a broader class of systems code, but also about the performance *side-effects* that a callee can have on a caller due to shared micro-architecture.

CFAR leverages two key ideas from prior work on performance interfaces, particularly PIX [30]. First, just like PIX (and Freud [63]), CFAR represents performance properties as programs that are both human-readable and executable. 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. PIX's two-phased approach has also been used by other recent work: `ltc` [47] uses it to provide visibility into the performance of hardware accelerators, while Performal [76] uses a similar approach to verify the performance of distributed systems. That said, CFAR's key technical contributions are the abstractions of the distillates and projections, which are specific to CFAR's focus on cache usage.

Using automated program analysis to reason about performance properties of systems code. Given the recent advances in automated program analysis techniques [27, 33, 37, 39, 64, 71], there is work that leverages such analysis to reason about various performance-related properties of systems code. Violet [28] uses it to find configuration bugs in large cloud applications, Bolt [31] and Castan [57] use it to analyze the latency of software network functions, and Clara [61] proposes using it to analyze the performance impact of offloading programs onto SmartNICs. However, in each of these cases, symbolic execution is coupled with an analysis framework that is specific to the property of interest. In CFAR, we use symbolic execution to extract all the information about how a piece of code uses memory, and we enable developers to write projectors that transform this information into the answers that the developers need.

Understanding the 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 unseen workloads. All prior tools we know of are limited to providing insights about the specific workloads that the tool was run on.

We drew significant inspiration from work in the 90s on abstract execution [40] and memory tracing [21]. Both these efforts aimed 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, so they sought to generate this trace on the fly instead. CFAR's distillate thus represents a generalized version of their work, and builds on advances in automated program analysis.

More recent work has focused on building better profilers [10, 17, 35, 43, 49, 59, 69] to help developers fix performance issues that are caused by poor cache utilization. Such systems involve a fundamental trade-off 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 [49]: It 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, it can account for and explain every single cache miss and—using a processor-accurate model—can approximate memory-access latencies. However, Memspy requires porting applications to its simulator, which can be a painstaking task. Additionally, its high performance overhead restricts it to profiling a limited number of input workloads. At the other end of the spectrum are profilers like DMon [35]: These work off-the-shelf for almost any systems code, and have low enough overhead to run continuously in production. Their downside is that they can only be used to monitor a specific subset of events and cannot provide the visibility that MemSpy does.

We see profilers as complementary to CFAR. Distillates 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 greater detail for specific, concrete inputs.

7 Conclusion

Developers need better *abstractions* to reason precisely about the expected performance behavior of their systems. Developers today are forced to manually inspect or profile the system *implementation* directly, which is both time-consuming and error-prone, since most systems today rely on a lot of third-party code. This is in contrast to how developers reason about functionality, where abstractions such as specifications, interfaces and documentation have been widely used for decades.

In this work, we focused on helping developers reason precisely about how systems code interacts with the underlying micro-architecture, specifically the CPU cache. We presented CFAR, a technique that introduces an abstraction that precisely captures what a piece of code does to the micro-architecture as a function of its inputs (the distillate) and provides a simple means of "querying" this abstraction, to help developers efficiently answer diverse questions about cache usage of their own, as well as third-party code, without having to delve into the code's details or run time-consuming benchmarks. We see CFAR as a key step towards augmenting systems with *performance interfaces* that describe the system's performance behavior in a manner that is simultaneously succinct, precise, and human-readable, just like semantic interfaces describe functionality.

We used CFAR to analyze different types of systems code and demonstrated that it can help developers identify performance bugs and security vulnerabilities, as well as understand the performance impact of using third-party code in their systems. CFAR's analysis completes in minutes, making it feasible to integrate CFAR into the software development cycle.

CFAR is publicly available as open-source software at [58].

8 Acknowledgements

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

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] Chipounov, V., Kuznetsov, V., and Candea, G. S2E: A platform for in-vivo multi-path analysis of software systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2011).

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

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

[18] 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).

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

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

[21] 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).

[22] 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).

[23] 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).

[24] 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).

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

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

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

[28] 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).

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

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

[31] 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).

[32] 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).

[33] 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).

[34] 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).

[35] 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).

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

[37] 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).

[38] 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).

[39] 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).

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

[41] 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).

[42] 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].

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

[44] 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).

[45] 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).

[46] 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).

[47] 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).

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

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

[50] 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).

[51] 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).

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

[53] 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).

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

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

[56] 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].

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

[58] CFAR project website. https://dslab.epfl.ch/research/perf, 2024.

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

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

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

[62] 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).

[63] 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).

[64] 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).

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

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

[67] 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).

[68] 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).

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

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

[71] 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).

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

[73] 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).

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

[75] 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).

[76] 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).

[77] 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).

[78] 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).

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