A Domain-Specific Language for Reconfigurable, Distributed Software Architecture

Henry Zhu
Department of Computer Science
University of Illinois Urbana-Champaign
Urbana, IL, USA

Junyong Zhao
Department of Computer Science
University of Arizona
Tucson, AZ, USA

Nik Sultana
Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA

***Abstract*—A program's architecture—how it organizes the invocation of application-specific logic—influences important program characteristics including its scalability and security. Architecture details are usually expressed in the same programming language as the rest of a program, and can be difficult to distinguish from non-architecture code. And once defined, architecture is difficult and risky to change because it couples tightly with application logic over time.**

We introduce C-Saw: an approach to express a software's architecture using a new embedded domain-specific language (EDSL) designed for that purpose. It *decouples* application-specific logic from architecture, making it easier to identify architectural details of software. C-Saw leverages three ideas: (i) introducing a new, formally-specified EDSL to separate an application's architecture description from its programming language; (ii) reducing architecture implementation to the definition and management of distributed key-value tables, and (iii) introducing an expressive state-management abstraction for distributed applications.

We describe a prototype implementation of C-Saw for C programs and use it to build end-to-end examples of expressing and changing the architecture of widely-used, third-party software. We evaluate this on Redis, cURL, and Suricata and find that C-Saw provides expressiveness and reusability, requires fewer lines of code when compared to directly using C to express architectural patterns, and imposes low performance overhead on typical workloads.

***Index Terms*—Key-Value Tables, Process Algebra, Coordination Language, Domain-Specific Language**

I. Introduction

Software's architecture describes its fundamental information-processing structure [1] and varies in its complexity. Examples of architecture include: a sequence of processing steps, a pipeline of concurrent stages, an event-handling system, a fan-out to worker instances, and a mix of these patterns [2].

The choice of architecture influences important software characteristics such as security [3] and performance [4]. For example, architecture affects how software can scale to meet demand by harnessing additional resources to distribute the load from computation, communication and storage demands.

Since both *architecture* and the *application-specific* logic are usually described using the same programming language, there is no language-level distinction between them, and no obstacle to them being tightly coupled over time. As a result, it can be difficult to alter one without affecting the other [5]. The blurring of architecture and logic complicates the implementation of important features that depend on architecture-level changes. Fig. 1 shows examples of such features which include caching and load-balancing.

As a result of architecture's poor visibility in source code and its coupling with non-architecture code, architecture-level changes are *high-friction*: they take effort, risk introducing bugs, and create a maintenance burden if the software diverges from an up-stream, canonical open-source version. One could avoid architecture-level change by designing an overly-general architecture to begin with, but this raises practitioners' red flags because it risks "premature optimization" [6], "creeping elegance" [7], and introducing a "bad smell" from needless complexity due to "speculative generality" [8]. Even then, general interfaces might not forestall the need for eventual revision since the software's requirements can evolve.

To avoid these problems, we need a low-friction method to express software's architecture. It needs to support a range of architecture patterns, be linguistically distinguished from application logic, and induce low overhead. New and existing software could then be adapted more easily to respond to new and changing needs that require architecture-level changes.

In this paper, we introduce C-Saw ("see-saw"): an approach to express a software's architecture using a new Embedded domain-specific language (DSL) designed for that purpose. C-Saw relies on distributed key-value tables to track both architecture-related state and application-logic state. These tables are managed by DSL expressions. The DSL is inlined into the application source-code and it is designed to work with existing software and languages—we prototyped this for the C language and developed usage examples involving widely-used, third-party applications.

The DSL can express a set of *architectures* that serve commonly-occurring *needs* such as those serviced by the examples in Fig. 1. These needs include: **(i)** *availability* through fail-over or replication; **(ii)** *manageability* through live migration or scale-out; **(iii)** *performance* through caching for latency, load-balancing for throughput, or object-size sharding for lower scheduling overhead; **(iv)** *lower resource cost* through scale-in or fusion of instances; **(v)** *security* through remote auditing.

The key idea in C-Saw involves decoupling an application's general architecture description from its application-specific

logic. C-Saw shrinks the scope of understanding and changing a software's architecture, thus lessening the effort and risk. The DSL has restricted expressiveness to limit unwanted behaviors. It provides a concise syntax and formal semantics to channel intuition into short and accurate architecture specifications.

We found that even intuitively-simple needs can have subtle and complex specifications, and this underscored the importance of using a specialized language for describing architectures. For example, we explored several formulations of fail-over logic that differed in redundancy, resourcing, and complexity. Another benefit of using the DSL is that architecture specifications are more *reusable* since they are decoupled from application-specific logic. We evaluate reuse of logic expressed using the DSL in our prototype.

C-Saw is inspired by coordination languages [9], Architectural Description Languages (ADLs) [10], and process algebrae [11], but it has important differences: (1) C-Saw is designed for use in existing software and tool-chains rather than necessitate a rewrite into a new language, and (2) Unlike process algebrae, C-Saw was designed for use in deployment contexts that do not typically allow an all-to-all communication model and higher-order channels between distributed components, in the interest of improved security and lower overheads. Also, more emphasis is placed on the interface to the host language which is used to describe application logic.

C-Saw's prototype uses libcompart [12] as a distributed runtime that coordinates logic across the program's architecture. It provides configuration, communication, and fault-handling support. Unlike microservices [13]–[15] and distributed OSs [16], C-Saw is focused on a language-level scope. Broader OS-level scoping is left for future research.

C-Saw's current design necessitates modification of the software's source code to derive subprograms that the DSL-expressed architecture will then interface with. Currently, this modification is done manually, and we evaluate its intrusiveness for the third-party systems to which we applied the C-Saw prototype. Automating this analysis and transformation is a separate research project and is left as future work.

This paper makes the following novel **contributions**:**(a)** A design (§III) that integrates C-Saw with existing C code-bases; **(b)** A formally-specified domain-specific language (§VI) to separate an application's architecture description from its programming language; **(c)** A diverse suite of DSL-encoded architectures (§V) covering all the examples shown in Fig. 1, all of which were implemented on third-party software (Redis, cURL, and Suricata). **(d)** An expressive state-management abstraction (§VII) that supports the DSL's C prototype to provide distributed synchronization. **(e)** A prototype implementation of C-Saw's DSL and state-management abstraction for C, and this prototype's evaluation (§VIII) for expressiveness, reusability, effort and performance on typical workloads.

The C-Saw prototype and evaluation suite are made freely available [17]. Additional usage examples and language details of C-Saw are provided in a technical report [18], including the formal semantics of C-Saw.

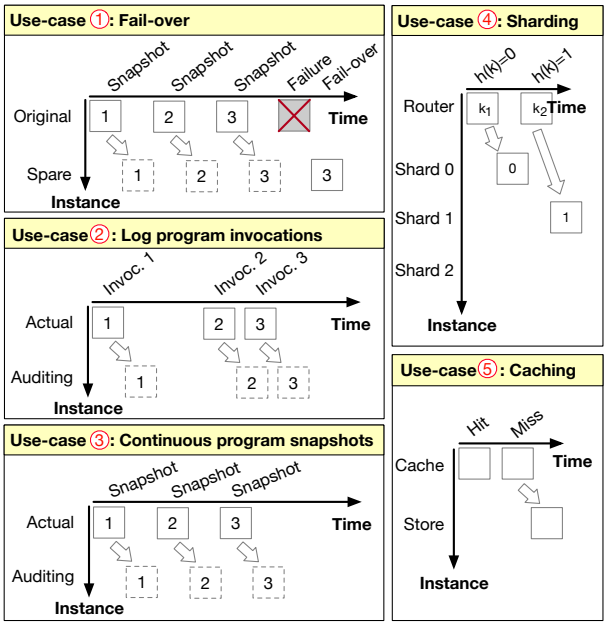


Figure 1: Behavioral sketches of architectural configurations that provide important application features. These diagrams give a simplified glimpse of architecture-related interactions over time. The x-axis shows events occurring over time, such as state-snapshots, application invocations, failures, or queries. The y-axis shows state synchronization occurring across specialized *instances* derived from the original software. Instances are components of the original application that capture distinct non-architectural features. Sketch ① describes fail-over, ② and ③ describe remote, integrity-preserving auditing of a local process (the first is *one-time* and the second is *continuously* running), ④ describes sharding, and ⑤ describes caching. These examples are detailed further in §II.

II. Software Architecture Use-cases

Features such as those in **(i)-(v)** from the previous section are typically cross-cutting, application-wide features that rely on architecture-level support. This section defines software architecture to make this paper self-contained. It also gives examples of architectural modifications of widely-used, open-source systems that are used to evaluate C-Saw later in the paper: cURL, Redis, and Suricata.

We define *architecture* to mean the code that defines the *delivery system* of *work* between other code that implements application logic. Architecture is generally dynamic: it does not only describe a structure of application-logic invocations, but describes how that structure responds to external or internal changes—such as changes in demand or in resource availability. Perry and Wolf [1] provide an excellent further discussion of software architecture with examples.

Changing a software's architecture can affect its *capacity to do work*, or *differentiate kinds of work from each other*. In Fig. 1, example ① increases work capacity across a failure mode, and example ② adds new work: capturing select state

for remote auditing.

Redis is a widely-used [19] NoSQL database [20] that is implemented as a single-threaded server [21]. We envisage three scenarios where architectural changes could benefit Redis, corresponding to examples ④, ⑤ and ① in Fig. 1. **1) Scaling:** Redis does not automatically scale with the number of CPU cores. It is scaled by manually starting more Redis instances or by using external harnesses such as Redis Cluster [22] but this relies on all-to-all TCP connections. It also relies on Redis Cluster or external tools [23] for sharding [24]. By internalizing this feature we can save overhead and resources. **2) Availability:** Redis employs a leader-follower system for replication [25]. The leader streams changes to the follower process. An architecture-level approach to providing this feature involves on-demand checkpointing of Redis—the architecture would serialize state from across an instance—and resuming Redis from a checkpoint. This approach also weakens the requirement for the leader and follower to be synchronously active. **3) Performance:** Redis instances are typically heavy users of memory, and by having a cache for frequently-accessed objects we can evaluate configurations that outperform a instance [26].

cURL [27] is a library and client for transferring data using a variety protocols. It is widely-used [28] and we envisage scenarios where its architecture is changed to support remote auditing—such as examples ② and ③ in Fig. 1. For example, this could be used on employer-provided machines or storage partitions as part of a security or compliance-checking policy for a company that allows Bring-Your-Own-Device (BYOD) [29]. Or for device owners to better track their smart consumer electronics [30]. We subdivide this scenario into two use-cases: The **first** captures program state at a key point of an invocation, such as at the start of the program. The **second** captures program state continuously, trading-off a higher runtime overhead to acquire more information. State is logged remotely to protect its integrity. Both of these configurations rely on a state-snapshotting feature, similar to that described for Redis above.

Suricata [31] is one of the three foremost systems used for network security monitoring [32]. It implements a graph-based abstraction for packet handling, reminiscent of Click [33]. Packet analysis and threat detection tasks are interconnected in this graph, which is executed on a multi-threaded abstraction of the underlying hardware. We envisage two scenarios for changing Suricata's architecture, corresponding to ① and ④ in Fig. 1. **1) Availability+Diagnostics:** At present, improving Suricata's availability requires external infrastructure to create a live Suricata replica that takes over in case of a crash. But whatever caused one Suricata instance to fail might cause its replica to fail too unless the cause was non-deterministic or resource-related. We can create a modified form of availability that involves continuously checkpointing Suricata state and resuming from the checkpoint in case of a crash. If the replica fails too, then we can use the checkpoint to reproduce the fault and understand it. To this end, we reuse the architectural pattern described earlier for fail-over in Redis, and interface it with Suricata's task graph. **2) Flow-level resourcing for performance:** We reuse the sharding logic from the earlier change to Redis' architecture, and use this to reserve resources for specific network flows identified as a 5-tuple [34]. This architectural configuration adds a policy layer on top of Suricata's allocation of cores to reserve some cores to process traffic of interest.

III. C-Saw

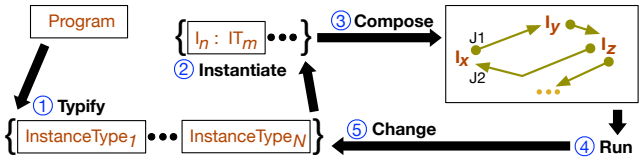


Figure 2: Workflow for C-Saw.

This section explains the workflow for adapting software to use C-Saw. The main steps in this workflow are shown in Fig. 2. The abstract description of the workflow in this section introduces the concepts of *junctions*, *instances*, and *instance types*. The sections that follow will provide concrete examples of applying C-Saw.

a) Adapting software to use C-Saw: Step ① in Fig. 2 involves **typifying** a program: this involves dividing it into different parts that can then be composed into a form described using the DSL. We call these parts *instance types* (or simply **types** for short). Each part implements a subset of the program's behavior that is related to a specific feature. For example, a *back-end* instance type includes the parts of a program that implement back-end behavior.

Types are then ② **instantiated** and named using the DSL to form *instances*. For example, a back-end instance *type* could be instantiated multiple times to form separate back-end *instances* for load-balancing. For another example, an application can implement fail-over between distributed replicas that instantiate the same type—this is expanded into a detailed use-case description later.

A type is instantiated, configured and connected using the DSL to form different architectures. The process of forming types depends on the feature-size granularity that is being sought. Our approach to apply C-Saw to third-party programs involved forming DSL expressions based on the coarse typifying of the programs to capture features that were described in §II. Our evaluation shows that even a coarse typification can accommodate (1) different adaptations to a program's architecture once C-Saw is used, and that (2) the same architectural description can be *reused* in different applications.

Types have one or more *junctions*: points in the control flow in which the instance type evaluates a DSL expression. Junctions are used to structure the coordination between instances.

Fig. 3 shows an example of how the architecture is made explicit for an abstract sequential program "$H_1; H_2$" and includes the C-Saw concepts that were described so far.

Instances can only communicate with each other through their type's junctions. For example, in Fig. 3 instances f and g (representing front-end and back-end instances respectively) can only send messages to a back-end instance through the junctions—and specifically by using the 'write', 'assert', and 'retract' statements in that example. This will be further illustrated using later examples.

Junctions and DSL expressions are **embedded** in the software's programming language. Introducing junctions into a program's source code involves inserting calls to C-Saw's API binding in the program's source code. The choice of where to introduce junctions, and how many to introduce, depends on the generality of the architecture sought: introducing more junctions creates more opportunities for architecture specification using DSL expressions.

Each junction has its unique name, a DSL expression, and a key-value (KV) table that stores state. The junction's behavior can be conditional on the table's contents. Instances communicate by making changes to each others' KV table when evaluating DSL expressions in junctions. Junctions can update their tables and those of other junctions through the evaluation of DSL expressions. All instances of a type share the same junction behavior but have a their own copy of the KV table. Using an analogy from Object-Oriented Programming, instance types are like classes and instances are like objects, but C-Saw does not support an inheritance hierarchy.

Instance types may have an arbitrary number of junctions, and each junction may communicate with an arbitrary number of other instances. Examples in the next sections will help make this clearer. Step ③ in Fig. 2 shows I_x having two junctions, J1 and J2, through which it exchanges KV updates with I_y and I_z respectively.

As will be shown in later examples, in addition to the architecture's structure and the communication between instances, DSL expressions also describe architecture-related logic that implements synchronization of KV-entries between instances, time-outs, retries, and fan-out and fan-in behavior. The DSL's constrained expressiveness makes it easier and safer to change its code than the general programming code in which it is embedded. The DSL supports limited recursion and it is not Turing complete.

b) Architecting software using C-Saw: Software architecture is realized using C-Saw through the ③ **composition** of instances using the C-Saw DSL, by defining the behavior of junctions.

Later, a typification can be ⑤ **changed** to support different junction behavior or granularity. For the C-Saw prototype, we evaluate the changing of DSL expressions in an application modified to use C-Saw without changing its typification, and the reuse of DSL expressions across applications.

c) Running software composed using C-Saw: In addition to the modifications to the program's source code to use C-Saw, the program's compilation is changed to link with a runtime system that interconnects C-Saw instances. The C-Saw prototype implementation uses libcompart [12]: a lightweight, portable runtime that provides channel abstractions for communication between instances. Its channels wrap OS-provided IPC, including TCP sockets and pipes. Each instance executes on the runtime. The runtime controls how instances interact with each other and undertakes message-passing between them, under the control of DSL expressions. Messages may contain serialized application data or control messages, as described in the next section. Serialization support for C data is described in §VII.

④ **Running** a program whose architecture is described using C-Saw involves starting a special instance that computes the "main" function. In turn, this can start other instances that form the program's architecture, as we saw in Fig. 3.

```
InstanceTypes = {τ_f, τ_g}
Instances = {f : τ_f, g : τ_g}
def main() ◀ start f(g) + start g(f)
def τ_f::junction(ḡ) ◀ ❶
  | init prop ¬Work
  | init data n
  ⌊H_1⌉; save(..., n);
  write(n, ḡ);
  assert [ḡ] Work; ❷
  wait [] ¬Work; ❸
def τ_g::junction(f̄) ◀
  | init prop ¬Work
  | init data n
  | guard Work ❹
  restore(n, ...);
  ⌊H_2⌉;
  retract [f̄] Work; ❺
```

Figure 3: Example in C-Saw DSL that typifies the program '$H_1; H_2$' into τ_f (instantiated as f) and τ_g (instantiated as g), and showing the special function 'main'. Each '**def**' in this example is a *junction* whose body is embedded in the host programming language. Taken together, these definitions describe the architecture of the program. A detailed explanation of this example's syntax is given in §IV.

IV. Architecture Descriptions in C-Saw

The DSL is designed to describe a broad set of architectures that meet various needs, including those in Fig. 1. This section outlines the language, which will be described in more detail in §VI after we present illustrative use-cases.

The example in Fig. 3 will be used to introduce the DSL. The DSL describes a set of running *instances* arranged into a topology that forms the software's architecture. Instances are derived from programs as described in §III and are declared in the set Instances. They are given a single type from InstanceTypes, as shown in Fig. 3. Both instances and types are given names by the software architect.

Instances communicate with each other through their *junctions*. Junctions are used to structure concurrent execution on distributed state, and the DSL serves to make their coordination and synchronization explicit. We will encounter examples

with multiple junctions, but in Fig. 3, both instances have a single junction, each called '*junction*'. A junction's execution is scheduled by the instance's application logic—e.g., a junction might be called to service a client request. Scheduling assumptions are made explicit using junction *guards*: on line ❶ of Fig. 3, junction '$\tau_f :: junction$' can be scheduled at any time, but line ❹ shows that '$\tau_g :: junction$' may only be scheduled when proposition Work is true.

The Work proposition is used to coordinate between the pair of junctions for rate-limiting, in this case by ensuring that only one instance is executing at a time. Instance f (the only instance of τ_f) asserts the proposition to instance g at ❷—this line updates the KV table of f and g, and is a form of communication between instances. Instance f awaits the retraction of Work at ❸—it awaits another instance to update its KV table. Retraction is done by g at ❺. In §V we will see how these language features are used to define more complex features like fault-tolerance and redundancy.

Junctions are examples of *definitions*. Definitions start with zero or more *declarations*. These are prefixed by the pipe symbol ("|"). For example, "|**init prop** ¬Work" declares the proposition Work and initializes it to false. "|**init data** n" declares a variable n. Variables and propositions form the junction's *state*, and their values are stored in the junction's KV table.

Definitions can reference code from the host language in which the DSL is embedded. We use H to range over host-language statements. The notation $\lfloor H \rceil \{\vec{V}\}$ encloses code in the host programming language and specifies writable state $\vec{V}$. This notation separates application logic from the architecture expression. Only junction state $\vec{V}$ may be *written to* by the host language statement H. Arbitrary junction state may be *read* by H. Fig. 3 abstracts host-language code as $\lfloor H_1 \rceil$ and $\lfloor H_2 \rceil$. Dropping the $\{\ldots\}$ means that neither H_1 nor H_2 may alter their containing junction's KV table.

V. Examples of C-Saw from Use-Cases

This section provides in-depth examples of using the DSL to implement architectural patterns from Fig. 1. These examples demonstrate the DSL's features before its syntax and semantics are described in the next section. The technical report [18] discusses additional and alternative designs that approach the problem differently to achieve a different trade-off.

A. Remote snapshots

Fig. 4 shows an implementation of example ② from Fig. 1: *one-time* remote snapshots. Act and Aud are both single-junction instances. Act forms part of the application, and Aud forms part of the remote logging system, reflecting the distributed architecture of the overall system. The logic mostly consists of a simple extension of Fig. 3.

This architecture can be reused for *continuous* remote snapshots ③ if we repeatedly invoke Act and Aud during a single execution of the overall system. This would repeatedly capture the *same* variables for logging; if we need to capture different variables during a program's lifetime then we would need additional instances or junctions. A multi-instance architecture example is given next.

```
InstanceTypes = {τ_Actual, τ_Auditing}
Instances = {Act : τ_Actual, Aud : τ_Auditing}
def main(t ❶) ◀ start Act(t) + start Aud(t)
def complain() ◀ ...
def τ_Actual :: (t) ◀
  | init prop ¬Work
  | init data n
  ⌊H1⌉; save(..., n);
  ⟨write(n, Aud);
   assert [Aud] Work;
   wait [] ¬Work;
  ⟩ ❷ otherwise[t] complain();
def τ_Auditing :: (t) ◀
  | init prop ¬Work
  | init prop ¬Retried
  | init data n
  | guard Work
  restore(n, ...);
  ⌊H2⌉;
  retract [] Retried; ❹
  case {❸
    Work ⇒
      retract [Act] Work otherwise[t]
        if ¬Retried then assert [] Retried;
        else complain();
      reconsider ❺
    otherwise ⇒ skip
  }
```

Figure 4: Remote snapshot example. This extends the example from Fig. 3 with failure-awareness and tolerance. The main differences from Fig. 3 are: ❶ accepting a timeout parameter, ❷ using this time-out for failure-awareness and for ❸ retry-based failure-tolerance. The first time this junction is scheduled, line ❹ is redundant since Retried is initialized to false, but line ❹ ensures that Retried is reset each time that position is reached, before the logic that follows it.

B. Sharding

We can implement sharding—example ④ in Fig. 1—through an architecture that routes queries to different back-ends according to an N-way partitioned query-space. This architecture could be repurposed to load-balance *computations* across an application rather than load-balance *storage* as is being done here.

Fig. 5 shows an implementation of sharding in the DSL. It features two new behaviors when compared to previous examples: **(i)** the DSL is interacting with the host language to obtain values such as hashes that cannot be computed in the DSL because of its restricted expressiveness, and **(ii)** abstracting over the number of backends, which is configuration parameter external to the DSL.

Function `Choose()` is defined in the host language and computes which backend to target, by allowing the tgt value to be written back to the DSL. We use the **idx** declaration syntax to allow the DSL to use externally-provided indices. The number N of back-ends is a compile-time configuration parameter. It affects the definition of Instances and line ❶ in Fig. 5.

Note that $\lfloor$`Choose()`;$\rceil\{\text{tgt}\}$ is sufficiently abstract to implement different types of sharding. The simplest sharding is key-based. Using C-Saw we implemented a Redis extension that provides a more complex, *feature-based* sharding based on object size to improve memory locality. We shard by performing a look-up on a custom table that maps keys to object sizes, and we quantize sizes into disjoint ranges: 0-4KB, 4KB-64KB, and >64KB. An improved sharding architecture is provided in a technical report [18] together with other usage examples of C-Saw.

```
InstanceTypes = {τFront, τBack}
Instances = {Fnt : τFront, Bck1 : τBack, ..., BckN : τBack}
def τFront :: (t) ◀
  | init prop ¬Work
  | init data n
  | idx tgt of {Bck1, ..., BckN} ❶
  ⌊Choose(); ❷⌉{tgt}; save(..., n);
  ⟨write(n, tgt ❸); assert [tgt] Work; wait [] ¬Work⟩
    otherwise[t] complain();
```

Figure 5: N-ary sharding over $\{Bck_1, \ldots, Bck_N\}$. This example builds on Fig. 4. The definition of τ_{Back} is omitted because it closely follows τ_{Auditing}. The **idx** syntax used at line ❶ to declare an index that can be updated by the host language. This update may occur on line ❷. An index over a set can also be used as a cursor as seen on line ❸, which resolves to Bck_{tgt}.

VI. Overview of the C-Saw DSL

After the illustrative examples presented in §V, this section describes the syntax and semantics of the C-Saw DSL. The formal semantics and a longer tutorial description of C-Saw are provided in a technical report [18].

The DSL syntax is summarized in Table I. To help explain the syntax, we will refer back to earlier examples.

a) Notation: We use $\vec{\cdot}$ to denote a collection of terms, and we use metavariables ranging over parameters p, instances ι, junctions γ, DSL-defined functions f, and named data n. Named data is stored in a Key-Value (KV) table that is local to each junction; the name is the key and the data is the value. The DSL is designed to clarify the logic governing the synchronization of these KV tables.

b) Host↔Junction sharing: Junctions share data between their KV table and the host language by using the 'save' and 'restore' primitives to move host data to and from the KV table. Data can be pushed to other junctions' tables using 'write' (see Fig. 3).

T	::=	break	|	next
	|	reconsider		
F	::=	P	|	false
	|	$\neg F$	|	$F_1 \wedge F_2$
	|	$F_1 \vee F_2$	|	$F_1 \longrightarrow F_2$
G	::=	F	|	$\gamma@F$
V	::=	$P \mid S \mid I$		
E	::=	$\lfloor H \rceil\{\vec{V}\}$	|	$\langle E' \rangle$
	|	$\langle\!\langle E' \rangle\!\rangle$	|	return
	|	write(γ, n)	|	wait $[\vec{n}]$ F
	|	save(n, $\vec{x}$)	|	restore($\vec{x}$, n)
	|	E_1; E_2	|	E_1 + E_2
	|	$\|\|_n \vec{E'}$	|	E_1 otherwise$[t]$ E_2
	|	stop ι	|	start ι $\gamma_1(\vec{p})\ldots$
	|	assert $[\gamma]$ P	|	retract $[\gamma]$ P
	|	$f(\vec{p})$	|	**verify** G
	|	skip	|	retry
	|	case {		
		$F \Rightarrow E'; T$		
		⋮ ⋮		
		otherwise $\Rightarrow E''$		
		}		

Table I: C-Saw DSL Syntax, described in §VI. Symbols: E are expressions; T are terminators used in case branches; F are propositional formulas and G are formulas that can be interpreted relative to a specific junction; P are user-defined propositions like Work in Fig. 3; and V ranges over different kinds of symbols: indices I, sets S, and propositions P.

c) Junction↔Junction synchronization and communication: Junctions synchronize parts of their KV tables by using the 'write' primitive to push records to another junction, or using 'assert' or 'retract' for propositional symbols. This is done for Work at line ❸ in Fig. 3. The 'wait' primitive blocks until a formula is true. It allows for specific records in the KV table to be updated by another instance.

d) Composition: This includes sequential (;) and parallel (+) composition. The special composition 'otherwise' is used for timed failure-handling. It is heavily used in examples (§V).

e) Control flow: All DSL code is terminating except for calls to host-language code. As described below, the DSL supports loops but they are unrolled at compile time. It also supports limited branching: 'reconsider' (line ❺ in Fig. 4) branches to the containing 'case' expression if a different match is made (i.e., if the propositions checked before the current guard have changed, or the current guard no longer applies) otherwise the expression fails. 'retry' branches back to the beginning of a junction and can only be invoked a fixed number of times within a single scheduling of a junction.

f) Blocks: Code blocks differ in what happens if a failure is encountered in the block. When using a transaction block $\langle\!\langle \vec{E} \rangle\!\rangle$, a failure results in a clean rollback of the KV table. $\langle \vec{E} \rangle$ does not rollback if E is not executed successfully—whatever changes have been made to the table up to that point will persist. This matters because we get different behavior depending on where we put 'otherwise'.

VII. Serialization framework

Data-structure serialization is an important supporting primitive for C-Saw's definitional approach to software architecture. Different architectures might require program data to flow differently across instances, and serialization provides the means to capture that data. In languages like C, it is difficult to serialize data structures using existing tools without either (i) limiting the types of data structures that can be expressed or (ii) burdening the programmer with custom serialization of data types. The challenges encountered when working with C datatypes include: a) void pointers which can represent any data type, b) arbitrary casting, and c) implicit size of memory objects that are managed by C's standard library allocator or by a custom allocator.

Various solutions have been devised over the years, including new DSLs [35] intended for implementing RPC, template-based schemes [36] for general types but requiring programmer effort to use their API, techniques to ensure memory safety [37], and specialized approaches for datatypes used in network protocols [38].

C-Saw builds on the C-strider [39] approach for the C language. C-strider implements a type-aware traversal of specific heap objects at runtime, and is guided by user-defined callbacks. It statically analyzes the source code to generate information about each type and generates serialization calls for each field of a type.

Unlike C-strider, C-Saw avoids having the programmer modify their source code or definitions—instead, they `#include` automatically-generated definitions by the C-Saw serializer. This serializer consists of a new libclang-based tool that analyzes C datatype definitions. To use the C-Saw serialization tool, the user specifies the type to serialize, answers some size-related questions if required by the tool, and the serialization file is produced. This tool sacrifices some flexibility but it has been sufficient for the complex datatypes involved in the third-party software that was used to evaluate C-Saw. Though the approach is general, our prototype only supports recursive datatypes up to a maximum, though configurable, recursion depth. For instance, linked lists are only serialized up to a maximum length. Though this might seem like a limitation, it protects against overflowing the serialization buffer. Supporting more flexible serialization through runtime analysis and buffer-resizing is left as future work.

VIII. Evaluation

We evaluate the **behavior of features implemented in C-Saw** (§VIII-A) by using reference workloads to measure the effect of DSL-implemented features (from Fig. 1) on performance and reliability. We evaluate **DSL cost and benefit** (§VIII-B) by measuring the effort of using the DSL when compared to using the host programming language directly; and the **performance overhead** (§VIII-C) of a C-Saw-based system compared to the original version.

These experiments target Redis v2.0.2, cURL v7.72.0-DEV, and Suricata v6.0.3. All these experiments were carried out on Ubuntu 16.04.7 LTS under Linux kernel version 4.4.0-198-generic, on an Intel i7-3770 CPU machine clocked at 3.4 GHz and with 8GB RAM. Experiments ran in separate VMs allocated 1GB RAM. Experiments were repeated 20 times and averaged and reported with their standard deviation, except for the cumulative distribution function (CDF) data which we obtained directly from `redis-benchmark`.

A. Behavior of features implemented in C-Saw

This section evaluates different, newly-added features to Redis and Suricata. Deployments usually build features *around* Redis and Suricata, but in this work we *internalize* important features (Checkpointing, Sharding, Caching) by using the DSL. For Redis we generated workloads by using `redis-benchmark` using its default parameters. For Suricata we used `bigFlows.pcap`, a public packet-capture benchmark that contains several flows from different applications [40].

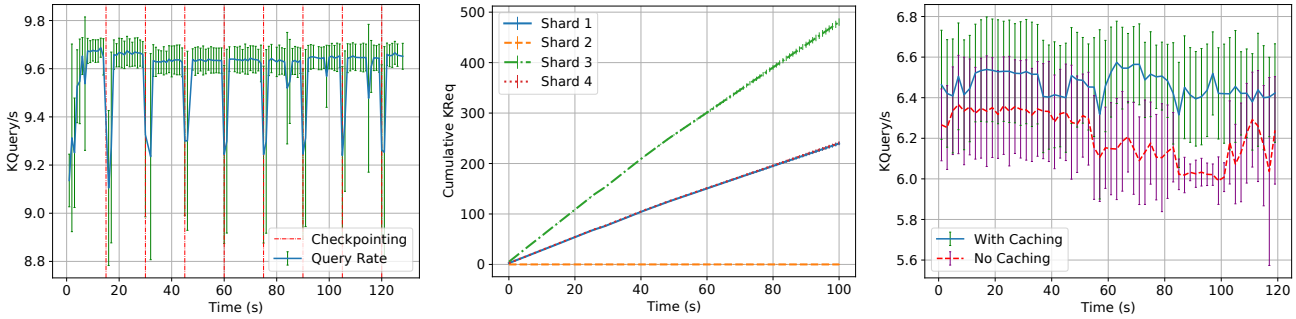
a) Checkpointing: Checkpointing is a building block for migration and roll-back of state. Fig. 6a shows the results for checkpointing Redis. Redis itself has a default crash handler but it does not checkpoint at intervals. While this handler improves availability, it does not minimize data loss. In this experiment we carry out checkpoints at 15-second intervals and simulate a Redis crash to observe its recovery. A crash is indicated by the vertical red line in the graph. Note that the y-axis starts at 8.0 to show the fine detail of the behavior—the dips in the graph are actually more subtle if we start the y-axis at 0. The same checkpointing logic was used in Suricata and the result is shown in Fig. 7a.

b) Sharding: We extended Redis using the DSL to implement two types of sharding, based on i) key and ii) object-size [41]. In both cases we sharded data into four classes, where each class is serviced by a separate back-end Redis instance. We subjected both types of sharding to even and uneven workloads. Uneven workloads place different pressure on different back-ends. Fig. 6b shows the results for sharding by key, which we hash using the djb2 hashing algorithm [42]. The graph shows the uneven behavior resulting from this workload; we confirmed that the ratio between shards matches that of the workload. The key-based sharding logic was adapted to implement packet-steering in Suricata, with the result shown in Fig. 7b.

c) Caching: We modified Redis to internalize a cache that is consulted before Redis' own look-up. We used a read-heavy workload to model a scenario where memory-burdened KV databases face a high skew in requests, modeling real-world scenarios [43], [44]. In our scenario, 90% of requests are directed at 10% of the entries. Fig. 6c shows response to this workload. Note that the y-axis starts at 5.6 to show the fine detail of the behavior; the gain from caching on this setup is around 200 queries per second (QPS).

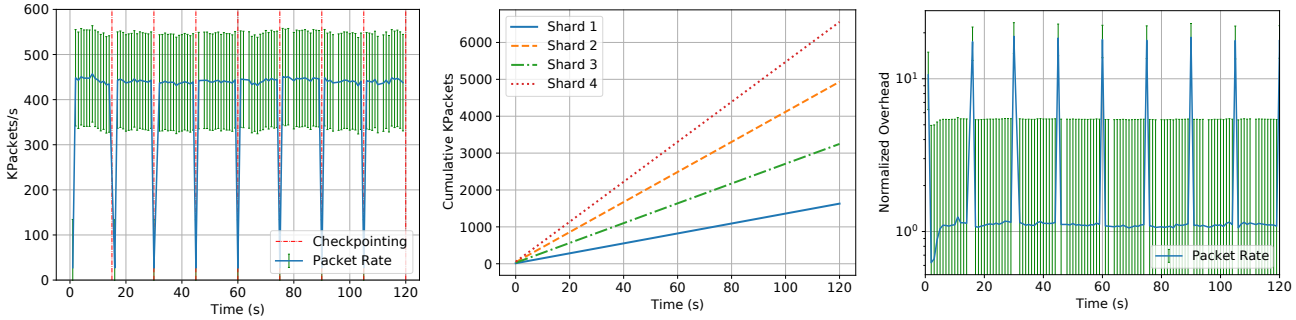
B. DSL cost and benefit

We measure cost and benefit by using lines of code (LoC) as a proxy metric for programmer effort. The Suricata and



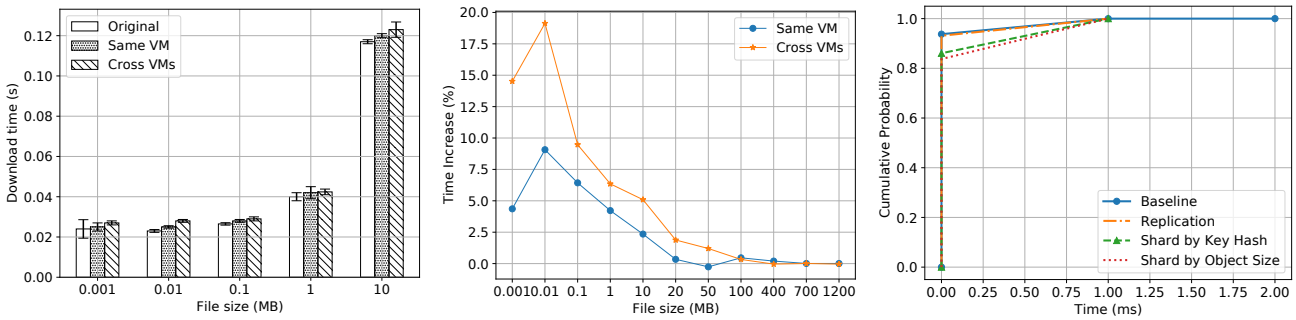
(a) Response of Query Rate to Checkpoints. (b) Cumulative requests sharded by size. (c) Effect of Caching on Query Rate.

Figure 6: Behavior of Redis reconfigurations. All three graphs show averaged results, and the bars show standard deviation.



(a) Response of Packet Rate to Checkpoints. (b) Cumulative requests sharded by 5-tuple. (c) Checkpointing Overhead.

Figure 7: Behavior of Suricata reconfigurations and normalized performance overhead of checkpointing.



(a) cURL performance (averaged). Bars indicate standard deviation.

(b) cURL overhead as percentage. Uses same data as Fig. 8a.

(c) Redis performance overhead (GET). 'Baseline' is unmodified Redis.

Figure 8: Overhead graphs for rearchitected software.

Redis codebases are at a comparable level of difficulty to understand—both are professional projects that are battle-tested by real-world usage. Since the DSL code is embedded as a C library, we give each LoC of DSL code the same weight as a LoC of C code for simplicity, although DSL code is arguably simpler than C syntax since it lacks various operators, unbounded loops, and pointers. The *cost* of using the DSL measures the code changes required to create instances and embed junctions. The *benefit* measures the LoC saved when extending an application by using the DSL compared to using C directly.

Feature	Lines Of Code (LoC) of C syntax			
	DSL in C	Redis(DSL)	Suricata(DSL)	Redis(C)
Checkpointing	79	7	44	332
Sharding	105	1	49	314
Caching	106	6	N/A	306

Table II: Effort (LoC) needed to support software extensions.

Table II shows the LoC needed to support different types of architecture-level features. **DSL in C** refers to the code generated by the DSL-to-C mapping that produces C code

that is decoupled from the application-specific logic. Once it is mapped in this way, the code can be used in a junction. **Redis(DSL)** and **Suricata(DSL)** refer to the number of lines edited in the source code to define the junction to host a DSL expression in Redis and Suricata for the different feature types. Defining a junction consists of packaging parameters and calling the "DSL in C" code. For Suricata most of the effort involved creating a new node in Suricata's pipeline that serves as a junction. **Redis(C)** is the LoC needed to rearchitecture directly in C. Redis(C) was developed without knowledge of the DSL, as a control experiment. It includes its own internal management system for communication and synchronization between different instances of Redis, which adds 195 lines to each feature.

a) Costs: The costs of using C-Saw involve typifying software and inserting junctions (§III). In Table II this cost is captured by **Redis(DSL)** and **Suricata(DSL)**. In addition to the per-architecture costs shown in Table II, there is a one-time cost for creating a junction that can be reused for different rearchitectures. For both Redis and Suricata we added a single junction consisting of 98 LoC for Redis and and 182 LoC for Suricata. The main function for Redis and Suricata received an additional 5 LoC and 4 LoC respectively to accommodate the junctions.

b) Benefits: The benefits of using C-Saw is three-fold: **i)** DSL expressions can be used across applications, which amortizes the effort of crafting a DSL expression. **ii)** Fewer code changes are needed—we can see this when comparing Redis(C) to the sum of "DSL in C" and Redis(DSL). **iii)** Support for serializing data that is exchanged between instances. The automatically-generated serialization code for the key and value structure used in Redis consists of 182 LoC. The generated serialization code for the packet structure used by Suricata consists of 2380 LoC.

C. Performance overhead

a) Suricata overhead: Fig. 7c shows the overhead of the checkpointing reconfiguration of Suricata normalized against the unmodified version. We see that overhead is usually less than 10% and spikes to around $19\times$ during checkpoint-restart-and-resume phases. The performance overhead of the sharding feature is around 60%, and it is graphed in §3 of the technical report [18].

b) cURL Overhead: We changed the architecture of cURL for remote auditing as described in §V. We generated two binaries: for the local and remote instances, respectively. The second binary is intended to receive progress updates from the first in order to audit it. We measured the overhead of executing this system when downloading differently-sized files from a dedicated machine, over 1GbE links on a research testbed. We ran two forms of this experiment: (i) placing both binaries in the same VM, and (ii) placing them in separate VMs to emulate separation between action and audit. Fig. 8b shows the overhead of cURL modified for remote auditing. Fig. 8a is based on the same numbers but presents them in absolute time, and shows standard deviation for more detail. The performance overhead for large files is less intelligible, so it has been relegated to §3 of the technical report [18].

Fig. 8c shows the response latencies when running GET operations on the original Redis and the three derivatives we developed. The graph shows that the overhead from our modifications are noticeable but low, except for "replication" which involves checkpointing and restarting Redis. While in many cases the average overhead is low, this experiment also features the longest tail latency albeit for a very small percentile. The results for SET are similar and shown in §3 of the technical report [18].

IX. Related work

Existing specification tools for software architecture, such as SysML [45] and arc42 [46], provide stand-alone descriptions of software architecture. In comparison, C-Saw specifications are embedded *inside* software. It will be interesting future research to look for a synthesis of C-Saw with SysML or arc42.

Split/Merge [47] involves classifying application state into two types depending on the state's scope: general state captures information across a whole application, while local state is scoped to invidual sessions or other units. C-Saw does not impose structure on state, and instead imposes structure on architecture patterns through a powerful DSL and primitives for state management.

C-Saw is inspired by research on process calculi [11] and on coordination languages [9]. Like these frameworks, C-Saw provides primitives for communication and coordination; but in contrast C-Saw provides a terser language that restricts flexibility at runtime. For example, C-Saw does not allow channel creation or passing, or an implicit global tuple-space. Further, it restricts mobility and interactions between processes. These restrictions simplify the runtime's implementation and the provisioning of resources, because they are not considered essential for the class of architectures we surveyed.

X. Conclusion

C-Saw is designed to "separate concerns" [48] between software architecture and its application-specific logic, and it provides a high-level DSL to declaratively compose this logic into an architecture. An architecture can then be reconfigured by editing its DSL expression.

Our main findings are: **(i)** When making them precise through C-Saw's formally-specified DSL, even simple behaviors such as those in Fig. 1 have subtlety and complexity. **(ii)** DSL expressions are reusable, and our prototype reused reconfiguration logic between Redis and Suricata.

Acknowledgments

This work was supported by a Google Research Award and the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-19-C-0106. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of funders.

REFERENCES

[1] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *SIGSOFT Softw. Eng. Notes*, vol. 17, no. 4, p. 40–52, Oct. 1992. [Online]. Available: https://doi.org/10.1145/141874.141884

[2] D. Garlan, "Software Architecture: A Travelogue," in *Future of Software Engineering Proceedings*, ser. FOSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 29–39. [Online]. Available: https://doi.org/10.1145/2593882.2593886

[3] J. I. Hong and J. A. Landay, "An Architecture for Privacy-Sensitive Ubiquitous Computing," in *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 177–189. [Online]. Available: https://doi.org/10.1145/990064.990087

[4] M. Welsh, D. Culler, and E. Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, p. 230–243, Oct. 2001. [Online]. Available: https://doi.org/10.1145/502059.502057

[5] N. Kew, *The Apache Modules Book: Application Development with Apache*. Prentice Hall Professional, 2007.

[6] D. E. Knuth, "Structured Programming with go to Statements," *ACM Comput. Surv.*, vol. 6, no. 4, p. 261–301, Dec. 1974. [Online]. Available: https://doi.org/10.1145/356635.356640

[7] B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Commun. ACM*, vol. 31, no. 11, p. 1268–1287, Nov. 1988. [Online]. Available: https://doi.org/10.1145/50087.50089

[8] M. Fowler, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[9] F. Arbab, M. M. Bonsangue, and F. S. de Boer, "A Coordination Language for Mobile Components," in *Proceedings of the 2000 ACM Symposium on Applied Computing - Volume 1*, ser. SAC '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 166–173. [Online]. Available: https://doi.org/10.1145/335603.335734

[10] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 3, p. 213–249, Jul. 1997. [Online]. Available: https://doi.org/10.1145/258077.258078

[11] R. Milner, *Communicating and mobile systems: the pi calculus*. Cambridge University Press, 1999.

[12] N. Sultana, H. Zhu, K. Zhong, Z. Zheng, R. Mao, D. Chauhan, S. Carrasquillo, J. Zhao, L. Shi, N. Vasilakis, and B. T. Loo, "Towards Practical Application-level Support for Privilege Separation," in *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*. ACM, 2022, pp. 71–87. [Online]. Available: https://doi.org/10.1145/3564625.3564664

[13] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Commun. ACM*, vol. 59, no. 5, p. 50–57, Apr. 2016. [Online]. Available: https://doi.org/10.1145/2890784

[14] C. Walls, *Spring Boot in Action*, 1st ed. USA: Manning Publications Co., 2016.

[15] I. Kuz, Y. Liu, I. Gorton, and G. Heiser, "CAmkES: A Component Model for Secure Microkernel-Based Embedded Systems," *J. Syst. Softw.*, vol. 80, no. 5, p. 687–699, May 2007. [Online]. Available: https://doi.org/10.1016/j.jss.2006.08.039

[16] S. J. Mullender, "Distributed Operating Systems," *ACM Comput. Surv.*, vol. 28, no. 1, p. 225–227, Mar. 1996. [Online]. Available: https://doi.org/10.1145/234313.234407

[17] "C-Saw repo," https://gitlab.com/pitchfork-project.

[18] H. Zhu, J. Zhao, and N. Sultana, "Semantics and further Use-Cases and Evaluation of the C-Saw language," Technical Report: repository.iit.edu, Mar. 2023.

[19] R. contributors, "Who's using redis?" https://redis.io/topics/whos-using-redis.

[20] R. Cattell, "Scalable SQL and NoSQL Data Stores," *SIGMOD Rec.*, vol. 39, no. 4, p. 12–27, May 2011. [Online]. Available: https://doi.org/10.1145/1978915.1978919

[21] "Redis Explained," https://architecturenotes.co/redis/.

[22] R. contributors, "Redis cluster specification," https://redis.io/topics/cluster-spec.

[23] T. Inc., "twemproxy," https://github.com/twitter/twemproxy.

[24] R. contributors, "Partitioning: how to split data among multiple redis instances," https://redis.io/topics/partitioning.

[25] ——, "Redis replication," https://redis.io/topics/replication.

[26] F. McSherry, M. Isard, and D. G. Murray, "Scalability! But at What COST?" in *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*, ser. HOTOS'15. USA: USENIX Association, 2015, p. 14.

[27] h. The cURL Contributors, "curl," https://github.com/curl/curl, 2021.

[28] ——, "Companies using curl in commercial environments," https://curl.se/docs/companies.html.

[29] A. Armando, G. Costa, and A. Merlo, "Bring Your Own Device, Securely," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 1852–1858. [Online]. Available: https://doi.org/10.1145/2480362.2480707

[30] G. Heiser, "Virtualizing Embedded Systems: Why Bother?" in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 901–905. [Online]. Available: https://doi.org/10.1145/2024724.2024925

[31] T. O. I. S. Foundation, "Suricata," https://suricata.io/.

[32] "Open source ids tools: Comparing suricata, snort, bro (zeek)," https://cybersecurity.att.com/blogs/security-essentials/open-source-intrusion-detection-tools-a-quick-overview.

[33] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, p. 263–297, Aug. 2000. [Online]. Available: https://doi.org/10.1145/354871.354874

[34] D. Zhou, Z. Yan, Y. Fu, and Z. Yao, "A Survey on Network Data Collection," *Journal of Network and Computer Applications*, vol. 116, pp. 9–23, 2018.

[35] "Protocol Buffers," https://developers.google.com/protocol-buffers/.

[36] "C serialization library," http://www.happyponyland.net/cserialization/readme.html.

[37] T. Ramananandro, A. Delignat-Lavaud, C. Fournet, N. Swamy, T. Chajed, N. Kobeissi, and J. Protzenko, "EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1465–1482. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/delignat-lavaud

[38] J. Bangert and N. Zeldovich, "Nail: A Practical Tool for Parsing and Generating Data Formats," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 615–628. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/bangert

[39] K. Saur, M. Hicks, and J. S. Foster, "C-strider: type-aware heap traversal for C," *Software: Practice and Experience*, vol. 46, no. 6, pp. 767–788, 2016. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2332

[40] "Sample Captures," suricata_sharding_vs_unmodified_diff.pdf.

[41] D. Didona and W. Zwaenepoel, "Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 79–94. [Online]. Available: https://www.usenix.org/conference/nsdi19/presentation/didona

[42] O. Yigit, "Hash functions," http://www.cse.yorku.ca/~oz/hash.html.

[43] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-Scale Key-Value Store," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, p. 53–64, Jun. 2012. [Online]. Available: https://doi.org/10.1145/2318857.2254766

[44] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at Twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 191–208. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/yang

[45] "OMG Systems Modeling Language," https://www.omgsysml.org/what-is-sysml.htm, 2022.

[46] G. Starke, "Documenting software architecture with arc42," https://www.innoq.com/en/blog/brief-introduction-to-arc42/, 2022.

[47] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. USA: USENIX Association, 2013, p. 227–240.