Automated Verification of Network Function Binaries

Solal Pirelli[+] Akvilė Valentukonytė[°*] Katerina Argyraki[+] George Candea[+]
[+]EPFL [°]Citrix Systems

Abstract

Formally verifying the correctness of software network functions (NFs) is necessary for network reliability, yet existing techniques require full source code and mandate the use of specific data structures.

We describe an automated technique to verify NF binaries, making verification usable by network operators even on proprietary code. To solve the key challenge of bridging the abstraction levels of NF implementations and specifications without special-casing a set of data structures, we observe that data structures used by NFs can be modeled as maps, and introduce a universal type to specify both NFs and their data structures, the "ghost map". In addition, we observe that the interactions between an NF and its environment are sufficient to infer control flow and types, removing the requirement for source code.

We implement our technique in Klint, a tool with which we verify, in minutes, that 7 NF binaries satisfy their specifications, without limiting developers' choices of data structures. The specifications are written in Python and use maps to model state. Klint can also verify an entire NF binary stack, all the way down to the NIC driver, using a minimal operating system. Operators can thus verify NF binaries, without source code or debug symbols, without requiring developers to use specific programming languages or data structures, and without trusting any software except Klint.

1 Introduction

Network operators are moving from hardware *network functions* to software ones for flexibility, but still deploying them as black boxes. Historically, network functions such as firewalls, NATs, and load balancers used special-purpose hardware for performance at the cost of flexibility, but this trade-off does not always make sense any more thanks to the performance of modern general-purpose hardware. Developers write software network functions using programming languages such as C or Rust and frameworks such as DPDK [15] or BPF [11, 33]. Marketplaces for distribution of software network functions are emerging [18], and most network functions on these marketplaces are proprietary and distributed in binary form.

Developers and operators currently test network functions hoping to catch bugs, but this is not enough especially in light of frequent software updates. For instance, network address translators from Microsoft [36], Linux [37], and Cisco [38] have had headline-causing bugs and vulnerabilities.

```
function Firewall_Check(packet, flow_table)
    if packet.device.is_internal then
        assert packet.is_forwarded
        if packet.flow not in flow_table then
            assert flow_table.was_full
    else
        if packet.is_forwarded then
            assert packet.flow in flow_table
```

Algorithm 1: Specification for a firewall. The firewall must remember flows exiting the network and must only allow packets in an existing flow to enter the network.

Operators need guarantees that the software network functions they deploy conform to specifications, instead of relying on tests. Consider the specification for a firewall in Algorithm 1, which will be our running example. This specification restricts what a firewall can do but does not state exactly how a firewall should be implemented. It also hides details irrelevant to operators such as packet parsers and policies to expire old flows. If an implementation could be *verified* to conform to this specification no matter what, an operator could deploy that implementation with confidence.

Developers need to provide proofs of functional correctness but do not always want to disclose their source code, meaning verification must be done on binaries. This requirement is driven by the way software is distributed to operators, but it also means developers do not need to restrict which programming languages they use nor worry about latent bugs in the tools they use such as compilers. A tool that can verify binaries can be used by operators regardless of whether they have access to source code and regardless of which language and toolchain developers used. Thus, even for open-source code, verifying binaries helps developers by enabling them to use any language and toolchain, even those considered experimental, since they can provide guarantees about the compiled binary to operators.

Developers currently cannot provide proofs (1) of functional correctness, (2) without source code, and (3) without verification expertise. They can use the Linux kernel's automated BPF verifier, but only for low-level properties such as memory safety. They can use Vigor [53] or Gravel [55] to automatically prove functional specifications, but both rely on typing information and thus require source code or an intermediate representation that can be reverse-engineered. Vigor and Gravel also limit developers to specific data structures; while developers could add new ones to the tools, they generally lack the verification expertise to do so. What remains is conventional testing, which can only show that a binary works in specific cases, not provide guarantees.

[*]Work done while at EPFL

Verification of data structures is out of scope for this paper. We assume that data structure code in binaries is clearly delineated and that it correctly implements its API and specification. Developers should then be able to write code using any data structure they want. A verification tool for network functions must understand the semantics of data structures through some form of contract, and automatically reason about their contents without requiring proof annotations. Developers who wish to provide maximal guarantees can manually verify their data structures, or use existing verified ones, but this should not be required for network function code to be automatically verifiable.

Our goal is to design a tool to automatically prove that a network function binary conforms to a specification, given data structures assumed to be correct. The inputs to this tool are a binary with explicit calls to data structure operations, contracts for these operations, and a specification such as a formalization of an RFC or IEEE standard. The output is a proof that the binary refines the specification or a counter-example that demonstrates otherwise.

Verifying arbitrary binaries is hard since type and control flow information are critical to verification but hard to obtain without debug symbols. However, network functions are not arbitrary. They commonly confine complex code to well-defined data structures, such as BPF maps [2], and only have a handful of well-defined interactions with their environment, such as transmitting packets and reading system time. These two observations lead to two ideas enabling us to verify network function binaries.

First, we use maps as a "universal" data structure to specify network functions and data structures. We define the abstract state of both network function specifications and data structure contracts in terms of maps from keys to values using a programming language such as Python. We call these maps *ghost maps* by analogy to "ghost variables" that exist only in proofs, not in implementations. Contracts can be written even for data structures that cannot be implemented in terms of maps, by using "for all" quantifiers to describe an operation's effects on the data structure in a declarative manner without describing its implementation. Verification is thus proving that the binary manipulates concrete data structures in a way that conforms to the manipulation of abstract maps in the specification, using contracts to match concrete operations with abstract ones.

Ghost maps enable efficient and automated invariant inference. Inferring invariants ensures automated analysis does not explore impossible program states, which would cause spurious verification failures. The sweeping simplification of maps enables our tool to reason about invariants across any data structures instead of limiting itself to specific ones. It also leads to the tool being simpler, as there is only one kind of data structure to reason about, the map. The tool translates data structure operations into map operations and infers invariants on the resulting maps.

Second, we infer typing and control flow information from environment interactions. Modeling network functions' environment precisely is feasible since it is small and well-defined. Typing information at the boundaries between a binary and its environment is enough, since specifications are defined in terms of environment interactions: maps and fundamental operations such as packet transmission.

We have built KLINT, a prototype of our technique. It takes in a binary without debug symbols, a Python specification of its semantics, and Python contracts for its data structures. The contracts only need to be written once per data structure implementation, as a more precise form of the documentation developers currently write. KLINT always terminates with either a proof or a counter-example, unless the processing of a single packet does not terminate, in which case KLINT fails after a user-defined limit. KLINT can verify an entire software stack using a minimal operating system if needed. KLINT can verify code written in different languages, such as C or Rust, on different frameworks, such as DPDK or BPF, and deployed in different contexts, such as virtual machines, containers, or raw hardware. Developers can use any data structure as long as it has Python contracts, even if the data structure implementation is not verified. The code of the verified network functions is *verification-agnostic*: only the standard programming practice of separating data structures from other code and documenting their specifications is necessary.

We use KLINT to verify network functions we write based on those from Vigor [53] and to verify existing BPF network functions such as Facebook's Katran [47]. KLINT can prove a range of properties, from full functional correctness according to a specification, to memory safety and crash freedom for functions without a high-level specification such as Katran. For instance, we extract a specification from IEEE 802.1D [30] for a bridge we write. We provide a detailed list of properties we verify using KLINT in Appendix A. Using KLINT gives operators guarantees about the correctness of any binary they deploy: either KLINT finds bugs, which can be reported to developers, or it finds no bugs, providing a guarantee the binary conforms to its specification.

In summary, our contributions are (1) a technique to reason about data structures and infer invariants based on the idea of "ghost maps" and (2) KLINT, a prototype that uses map-based reasoning to automatically and efficiently verify network function binaries that use trusted data structures with map-based contracts. We use KLINT to verify the functional correctness of 7 binaries, 6 written in C and 1 in Rust, in minutes. These verified binaries run faster than previous verified ones thanks to relaxing restrictions imposed by previous work. We also verify the memory safety and crash freedom of 5 existing BPF network functions using KLINT, though we have no specifications for them.

The code for KLINT and our network functions is publicly available at https://github.com/dslab-epfl/klint.

2 Design Insights

We formalize what we mean by a "network function" in §2.1, explain how environment interactions are enough to infer typing and control flow information of binaries in §2.2, and describe how we bridge the gap between implementations and specifications through the indirection of maps in §2.3.

2.1 Network functions we target

We restrict ourselves to network functions that use mutable state to process packets and contain two phases: initialization and packet processing. When processing a packet, network functions decide whether to transmit packets in response and what data to transmit based on state and on the packet's contents and metadata, such as its length. We assume that network functions are single-threaded, and that they only execute code when initializing and receiving packets. We do not support timers as triggers to run code; they must instead be checked during packet processing.

To create, read, and update state, network functions acquire and use data structure *capabilities* through an environment, such as the DPDK [15] framework. These capabilities correspond to opaque pointers in C, preventing network functions from directly accessing data structure internals.

Acquiring capabilities is only allowed during initialization, and may fail due to external factors, for instance running out of memory. Using capabilities is allowed at any time, and may only fail through incorrect use, for instance indexing arrays out of bounds. Packet processing may also use other environment calls that cannot fail from external factors, such as obtaining the current time.

Our assumptions about data structures correspond to good programming practices: developers should use data structures in a modular fashion for maintainability, and network functions should not allocate memory while processing packets, to avoid performance issues and out-of-memory errors that could allow for denial-of-service attacks.

We represent our model of a network function in Figure 1. This is only a formalization of what developers already do, not a proposal of a new model.

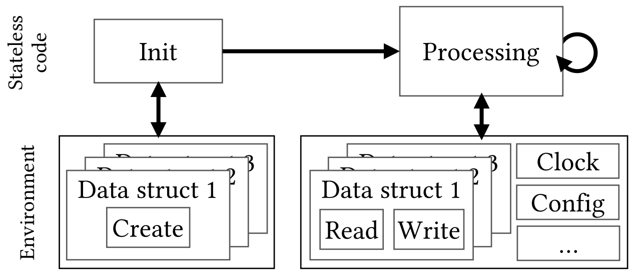


Figure 1. Network function components: initialization allocates state, packet processing reads and writes state and may also use other utilities. Arrows represent control flow.

2.2 Information from environment interactions

The first limitation of previous work that we overcome is the need for source code, thanks to the insight that environment interactions are sufficient to infer all necessary information about network function code, i.e., types and control flow.

As we described in §2.1, the only way for a program to hold state and interact with the external world is through its environment, such as displaying information or receiving a packet from the network. That is, we define the environment to be low-level enough that programs can be modeled as pure functions that compose environment interactions.

A tool can precisely infer all possible behaviors of a program by replacing the program's environment with one that can exhibit any allowed environment behavior, not a specific one. This verification-only environment can be written by hand or inferred from machine-readable documentation.

Writing an exhaustive model of the environment is not feasible for general-purpose software because its environment is large and ill-defined. There is no formal definition of large environments such as operating systems, and documentation often omits implementation details of high-level operations that programs may rely on in practice.

However, a complete environment description is feasible for network functions, because their environments are small and well-defined. Formally defining all environment interactions is manual work but must only be done once by an environment's developers, and the operations are low-level enough that emulating all possible implementation-specific behaviors is feasible. Having a small and well-defined environment also means that its modules can be formally verified manually if its developers choose to do so, enabling verification of the entire software stack.

For instance, our example firewall calls a data structure library to perform a lookup in its flow table and calls a networking framework such as DPDK to forward or drop the packet based on the lookup result. Both the data structures and the networking framework are part of the environment, and both can be precisely replaced by a tool as long as the network function uses dynamic linking.

Environment interactions can be observed on binaries without loss of information, since the tool knows the number of parameters and their types. For instance, when a binary calls DPDK's packet transmission function, the tool uses the current architecture's calling convention, which is known, and the data types in DPDK's headers, which are publicly available, to extract the arguments to the transmission function from the current machine state, such as the packet buffer and metadata. Thus, source code is not necessary for automated software network function verification.

The control flow information necessary for verification can be similarly extracted by observing which branches are taken or not between environment interactions, which produces an "unrolled" version of the true control flow graph.

2.3 Using maps to bridge the gap

Verifying the correctness of a network function that satisfies the definition in §2.1 consists of showing that the semantics of the code, obtained from environment interactions as described in §2.2, match the semantics of the specification. However, there is a gap between the variety of data structures that exist in the code and the restricted set of abstractions that any given tool can reason about, as we illustrate in Figure 2.

For instance, our firewall may use a "least recently used" data structure to keep track of which packet flows should be expired due to a lack of activity, with operations such as "add a new item" and "remove and return the oldest item". The firewall may also combine the use of this data structure with other structures, such as a map tracking per-flow statistics or a configurable set of ports that are open to the external world at all times.

Previous work proposed two ways to bridge the gap between implementation and specification: Vigor [53] requires the use of specific data structures in both implementation and specification, while Gravel [55] requires the use of specific data structures that it knows how to model in terms of high-level operations that can be used in specifications.

Vigor imposes two constraints to verify our firewall. First, the firewall must be written using data structures that Vigor knows about, either by modifying the firewall's code to only use Vigor data structures or by modifying Vigor itself to handle new data structures, including proof annotations for invariants that can hold across these data structures such as "all flows in the LRU are also in the map". Second, the firewall specification must be written in terms of the same data structures used in the implementation. These two constraints limit both developers and operators. Developers must restrict themselves to specific data structures or learn verification techniques to add new ones, and operators must learn the semantics of the data structures used in an implementation to understand its specification.

Gravel removes the second limitation: it translates data structure operations to a small set of high-level operations for use in specifications, thus operators do not need to learn all data structure semantics.

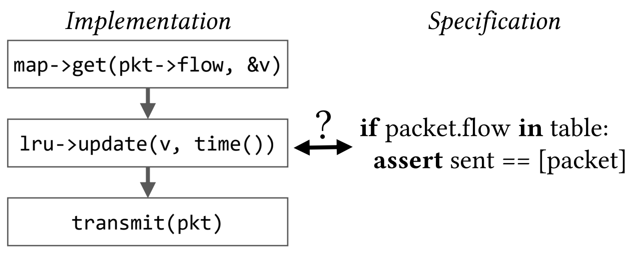


Figure 2. There is a gap between the abstractions used in the implementation and the specification.

However, the restriction on developers remains. Developers must either modify the firewall's code to only use existing Gravel data structures or modify Gravel itself to handle new data structures.

Fundamentally, tools that handle a specific set of data structures do not scale. Adding support for a new data structure requires not only encoding the data structure's operations, but also its interactions with other data structures, such as invariants that can exist across structures. Even if a tool is limited to invariants across two data structures, adding the Nth structure requires adding N kinds of invariants, one for each data structure including the new one.

We introduce a level of conceptual indirection: we express the semantics of both data structures and specifications in terms of one data structure, the *map*. Operations on data structures such as arrays, hash tables, longest-prefix-match tables, and port allocators can be defined in terms of map operations, regardless of how they are implemented. Verification tools can use contracts to translate any data structure operation into map operations, which become the only kind of operation the tool needs to reason about for invariant inference and verification.

We refer to such maps as *ghost maps*, by analogy to "ghost variables" which are variables only used in proofs. We present an example of contracts for a "least recently used" data structure in Listing 1. Using this contract, a verification tool can translate the LRU semantics into maps, and thus does not need special knowledge of what an LRU data structure is, only knowledge of maps. Importantly, contracts can be declarative, not just imperative. Ghost maps can define even data structures that cannot be implemented using maps thanks to the "for all" quantifier. `LRU_expire`'s contract only describes *what* it does, not *how*: the returned value is the oldest, but the contract does not need to explain how this value is found. We explain how a verification tool can efficiently reason with maps in §3.

This sweeping simplification also makes invariant inference easier since there is only one kind of data structure, as we show in §4.

```
# struct LRU;
  LRU = namedtuple('LRU', ['items'])
# void LRU_add(struct LRU* lru,
#              void* item, int age);
  assert item not in lru.items
  lru.items[item] = age
# void* LRU_expire(struct LRU* lru);
  age = lru.items[result]
  assume(lru.items.forall(lambda k, v: v <= age))
  lru.items.remove(result)
```

Listing 1. Contracts defined using maps, in Python, for a "least recently used" data structure, in C. `result` is the return value from `LRU_expire`.

3 Ghost Maps

To verify network functions automatically, we use *symbolic execution* to analyze all their paths, which we summarize in §3.1. Since data structure implementations have too many paths to enumerate, we abstract them using *contracts* instead, to obtain paths such as "key found" and "key not found" instead of one path per cell in which a key might be.

We introduce ghost maps as a "vocabulary" in contracts to describe the semantics of data structures and network functions to both verification tools and human readers. Ghost maps only exist within contracts, not implementations.

We describe our goals for ghost maps in §3.2, our proposed representation of ghost maps in §3.3, and our proposed translation of their operations into logical formulas in §3.4.

3.1 Symbolic execution background

A *symbolic execution engine* executes code with *symbols* as inputs instead of concrete values. Whenever it encounters a branch on a symbolic condition, it explores both alternatives, remembering the choices it made in a *path constraint*. This leads to a set of *paths*, which are sequences of choices that each represent one possible program execution. For instance, instead of executing an "absolute value" operation on -5 and obtaining 5, a symbolic execution engine will execute it on α and obtain two paths: one with path constraint $\alpha \geq 0$ and result α, and one with path constraint $\alpha < 0$ and result $-\alpha$.

Some code has too many paths to explore in reasonable time. For instance, consider looking for a value in an array. The value could be in the first position, or the second, or the third, and so on, until the array length. The number of paths is limited by the array length, which could be, e.g., 2^{32}. If the code which looked up the value then looks up another value, each path resulting from the first lookup will lead to new paths for the second lookup, squaring the total number of paths. This problem is known as *path explosion*. While the number of paths can sometimes be reduced by merging related paths at the expense of producing more complex constraints [29], the paths to be merged must still be partially explored, which does not solve path explosion.

If the number of paths in a program is "reasonably" small, a symbolic execution engine can exhaustively enumerate them and verify the program by verifying that each path satisfies the program's specification.

Figure 3. Ghost maps are more expressive than bitvectors while remaining decidable, unlike more powerful theories such as arrays with universal quantifiers.

$$
\begin{aligned}
length(M) &\to Int \\
get(M, K) &\to V \mid None \\
set(M, K, V) &\to M' \\
remove(M, K) &\to M' \\
forall\big(M, \lambda(k, v) \to Bool\big) &\to Bool
\end{aligned}
$$

Listing 1. Ghost map operations. M and M' are maps; K, V are keys and values. *Int* denotes bitvector-based integers, and *Bool* Booleans. *None* is the lack of value.

3.2 Expressivity, decidability, and completeness

We propose the ghost map abstraction in Listing 1 which is expressive enough to define data structures and network functions, while still enabling a tool to reason in a decidable, sound, and "complete enough" manner. We describe each of these properties next.

Ghost maps are *expressive* enough to abstract the data structures we care about. Simpler abstractions require too much detail in contracts to be practical for either humans or tools. For instance, representing a hash table as a sequence of 0s and 1s is possible but impractical. We are concerned with data structures used in network functions, such as hash tables and port allocators, thus we limit our vocabulary to what they need, not to all possible code.

However, expressiveness is at odds with *decidability*. Symbolic execution engines use a solver to tell whether a logical formula is *satisfiable*, i.e., whether there exists an assignment of variables such that the formula holds. For instance, if "the firewall's variables, given the firewall's constraints, violate its specification" is satisfiable, then the assignment of variables is a counter-example to the firewall's correctness. Logical formulas are written using *theories*, which are the "vocabulary" of solvers. Some theories are *decidable*, meaning that a correct solver will always return a correct answer. Some are not, meaning that the answer may be "unknown" instead of yes or no. Even with decidable theories, "unknown" may be returned if the solver is given a timeout and cannot find an answer in time.

Verification tools must be *sound* and as *complete* as possible. A tool is sound if it verifies *only* correct programs. A tool is complete if it verifies *all* correct programs. Due to the Halting Problem [49], verification of general-purpose programming languages must be incomplete, thus the goal is to verify "interesting" correct programs, i.e., those that humans actually write, even if some contrived theoretical examples cannot be verified.

Ghost maps are an intermediate step between quantifier-free bit vectors, which are decidable due to their finite size but not expressive enough, and arrays with universal quantifiers, which are more than expressive enough but undecidable, as we illustrate in Figure 3.

3.3 Representing ghost maps

To remain as decidable as the theory of bit vectors while offering more expressivity, we present a translation of ghost maps to bit vectors in the context of symbolic execution. Notably, ghost maps' "for all" quantifier, which enables non-imperative contracts for data structures that cannot be implemented with maps, can be translated *without* using universal quantifiers. Ghost maps can be more expressive despite being translated to bit vectors because the symbolic execution engine internally uses data structures, such as lists, to build the logical formulas it sends to the solver.

We expect the code to manipulate maps of large size, but to only interact with a small number of items in any given map. This is true of network functions, which by nature only perform a small number of operations for each packet to remain within their performance budget.

Tracking *known* and *unknown* items separately is our key insight to handle maps of arbitrary size. That is, none of the map operations require "forking" the current path, and the engine handles map operations in a time linear in the number of known items, not total items.

Thus, instead of keeping track of every item in every map, the engine only needs to track the specific items that are explicitly used in one iteration of the network function's packet-processing loop. Other items are "summarized" into a single pseudo-item that tracks their constraints, such as "all unknown values are non-zero".

This scheme naturally enables ghost map lengths to be symbolic, since their actual size does not matter as much as the number of known items. A verification tool can thus represent maps whose length is determined by configuration parameters using a symbolic configuration, instead of verifying only one specific configuration as is for instance done in Vigor [53]. The tool can thus catch all bugs that only occur in specific configurations, such as the maximum capacity of the firewall's flow table being zero, without requiring developers to think of which configurations to try.

Counter-intuitively, the engine must mutate its internal representation of maps during read-only operations on maps. Known items must include those that have been retrieved from the map, even if they have not explicitly been set before. For instance, consider the following:

$$get(M, K_1) \rightarrow V_1$$
$$get(M, K_2) \rightarrow V_2$$

If $K_1 = K_2$, then $V_1 = V_2$ by definition. But the engine must remember the first *get* in some way in order to guarantee the implication, and it cannot store high-level map operations in the path constraint, thus it must modify its internal representation of M. From an outside perspective, M has not changed, but internally the engine must remember this *get*.

We informally describe each operation first, then add additional details to handle subtleties, then provide a formal algorithm for the core *get* operation.

The engine tracks each map's length explicitly.

Known items are triples: (key, value, presence bit). If the presence bit is *false*, the value is ignored and the key is considered absent from the map. *None* only exists conceptually; the theory of bit vectors cannot represent it.

Known items may be redundant, but their values and presence bits must match if their keys match. This is because their keys may be symbolic, thus the engine cannot know for sure whether two items have equal keys.

Unknown items are represented by an *invariant* that they all satisfy. The unknown items invariant only concerns unknown items. Known items need not satisfy it. For instance, a map may have as invariant "all unknown values are non-zero" and two known items, $(K_1, 0, \mathit{true})$, $(K_2, 1, \alpha)$. The first known item is definitely present, whereas the second may or may not be present depending on the value of α.

The unknown items invariant is represented as a formula on a special *unknown item*, unique to each map. For instance, the non-zero invariant example is represented as $UP_M \Rightarrow UV_M \neq 0$, where UP_M and UV_M are the presence bit and value of M's unknown item. Including the presence bit in the invariant allows it to be constrained for cases such as arrays. For instance, a zero-based array of length L described using a ghost map A would have $UP_A = (0 \leq UK_A < L)$ as part of its unknown items invariant, indicating that keys are in A if and only if they are between 0 and L, matching the semantics of array indexing in languages such as C.

3.4 Translating ghost map operations

We use $ITE(c, t, f)$ to denote "if c then t else f", and *fresh* to mean a symbol that was not used before, i.e., that is not constrained in any way. "Applying" the unknown invariant of a map to an item means substituting the map's unknown item for the item.

length(M): Return the map's length, which the engine tracks explicitly.

get(M, K): Create a fresh tuple (V, P). Add (K, V, P) to the map's known items. Add constraints to the current path to encode that, within the map, (1) if K matches a known item then so do V and P, (2) if K does not match any known item then the unknown items invariant applies on (K, V, P), and (3) the number of unique known items cannot exceed its length. Return (V, P), which encodes "if P then V else *None*".

set(M, K, V): Let $(\_, P) = get(M, K)$. Return a new map whose length is $length(M) + ITE(P, 0, 1)$, whose known items are $\big(K', ITE(K = K', V, V'), P' \vee (K = K')\big)$ for each known item (K', V', P') in M, plus (K, V, true), and whose unknown items invariant is the same. That is, (1) the length only grows if K was not in M, and (2) known items must match if they are redundant.

remove(M, K): The opposite of *set*, i.e., the length change is $ITE(P, -1, 0)$, the new known item is (K, V, false), where V is any arbitrary value, and the known items are changed to $\big(K', ITE(K = K', V, V'), P' \wedge (K \neq K')\big)$.

forall(M, F): The result is *true* iff $P \Rightarrow F(K,V)$ for each known item (K,V,P) in the map and $UP_M \Rightarrow F(UK_M, UV_M)$ for the unknown item of the map. That is, the result indicates whether the predicate holds on known items and on unknown items. Add to the map's invariant that if the result of this operation is *true*, then the predicate holds on unknown items if their presence bit is *true*, ensuring that even if the result is not yet known, it will be consistently applied in the future. If the result is false, future items are not constrained, which is sound but not complete.

***Layers* are necessary to handle dependencies** that arise when code uses multiple versions of a map at once. Consider:

$$set\,(M, K, V) \longrightarrow M' \quad \begin{array}{l} get\,(M, K') \longrightarrow V' \\ get\,(M', K') \longrightarrow V'' \end{array}$$

If $K \neq K'$, then V' and V'' must be the same, since only the value associated with K is affected by the *set*. However, the representation described earlier cannot guarantee this, because the known items of M and M' are independent.

To handle this issue, *set* and *remove* return layers instead of entirely new maps, as the example in Figure 4 illustrates. That is, they return a map which includes the newly added or removed item among its known items, but which links to the previous map for the other known items, transforming them when necessary, instead of evaluating the new known items at creation time. Items and invariants created by *get* and *forall* are always added to the bottom-most layer. Map layers also share the unknown items invariant, and the associated "unknown item", of their base map.

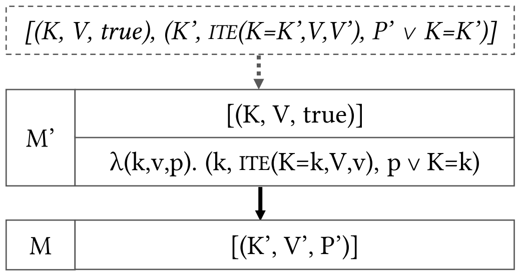
Thus, in our example above, the known items "seen" by the second *get* operation include the result of the first one, and thus the second *get* will return a result that lets a tool prove $K \neq K' \Rightarrow V' = V''$.

Invariant recursion must be explicitly handled to avoid infinite recursion when two maps' invariants refer to each other. For instance, a bi-directional map may be represented as two maps whose contents are inverses: for each key-value pair (K,V) in map M_1, there is a pair (V,K) in map M_2, and vice versa. The invariants are:

$$forall\big(M_1, \lambda(k,v).\ get(M_2, v) = k\big)$$
$$forall\big(M_2, \lambda(k,v).\ get(M_1, v) = k\big)$$

Consider what would happen using the representation described earlier when the engine handles $get(M_1, K)$ for some K. As part of adding the invariant on the newly-known item of M_1 to the path constraint, the engine will call $get(M_2, V)$ with the fresh V from the original *get*. As part of adding the invariant on the newly-known item of M_2 to the path constraint, the engine will call $get(M_1, V')$ with the fresh V' from the second *get*. The engine then calls *get* on M_2, and so on, leading to infinite recursion.

Solving this issue requires the engine to recognize that in the third *get* call, the V' argument is equal to K, which it knows from the first *get* call. The result should thus be the existing V, not some fresh V''.



[(K, V, true), (K', ITE(K=K',V,V'), P' ∨ K=K')]
M'
[(K, V, true)]
λ(k,v,p). (k, ITE(K=k,V,v), p ∨ K=k)
M
[(K', V', P')]

Figure 4. Example of a *set* layer M' on top of a map M, and the resulting known items of M'.

The engine thus tracks a *condition* and a *value hint* during ghost map operations, which are set when handling invariants and used to stop recursion when handling *get*.

When handling a map invariant of the form *get*(...) = ..., the engine adds to the condition the presence bit given as argument to the invariant and sets the value hint to the value expected by the invariant. These changes are reverted when the engine is finished handling the invariant.

$get(M, K)$ needs two changes at the start: First, if K cannot be different from an existing item's key assuming the condition holds, return that item's value and presence bit. Second, after creating the fresh V, if there is a condition, add "the condition implies $V = value\_hint$" to the path constraint.

Applying this logic to our example solves the issue. When handling $get\,(M_2, V)$, the value hint is K and the condition is P, both from the newly-known item of M_1. When M_2's invariant calls $get\,(M_1, V')$, the path constraint contains $P \Rightarrow V' = K$, thus the *get* on M_1 will start by checking whether its known item (K, V, P)'s key is equal to V' assuming P, which it is, and return (V, P), ending the recursion.

This strategy avoids recursion in common cases such as our example of maps with reciprocal keys and values, but it is not complete, as the engine may recurse infinitely. For instance, "M_1 has $min(K-1, 0)$ for all K in M_2, and M_2 has $min(K-1, 0)$ for all K in M_1" could hold, but will lead to infinite recursion in the engine given our implementation.

We present the final *get* algorithm, which is the core of our ghost map technique, in Appendix B. Our technique is decidable and expressive but not complete, unlike prior work that focused on completeness at the expense of expressiveness [3, 13]. Our technique enables a symbolic execution engine to translate ghost map operations into formulas on quantifier-free bit vectors. This enables the engine to bypass the path explosion caused by data structure code by executing the code's contract instead.

The universal "forall" quantifier on ghost maps allows contracts even for operations that cannot be implemented using maps by describing *what* they do instead of *how*. However, we believe some data structures are not a good fit for ghost maps, specifically ordered ones. While queues and stacks can be viewed as maps from indexes to elements, the resulting contracts are unlikely to be conducive to invariant inference.

4 Invariant inference

Handling data structures by translating them to ghost maps is not sufficient for automated verification. Developers use data structures in well-defined patterns, named *invariants*, but these patterns are not always explicit in the code.

Verification tools must *infer* such invariants to avoid failures. When executing a contract instead of an implementation, the tool must have enough information to prove the contract's precondition. Previous tools bypassed this problem by special-casing data structures and their invariants.

The ghost maps representation we proposed in §3 enables tools to infer invariants without special-casing templates.

Consider these motivating examples in pseudo-code:

```
if packet.flow in table:
  statistics.increment(packet.flow)
```

If `increment`'s contract requires the item to be in `statistics`, symbolic execution will fail if it cannot prove this fact.

```
device = destinations.get(packet.flow)
transmit_packet(packet, device)
```

If `transmit_packet`'s contract requires the device to exist, symbolic execution will fail if it cannot prove this fact.

```
if not items.full:
  items.add(x)
  metadata.add(x, y)
```

If `add`'s contract requires free space, symbolic execution will fail if it cannot prove `not metadata.full` in the second `add` call.

All three examples could be bugs depending on how the data structures they deal with are updated. If the first example is in code that always adds `packet.flow` to both `statistics` and `table`, the code is valid. If the second example always puts a device known to be valid in `devices`, such as the incoming device of a packet, the code is valid. If the third example is the only occurrence of `add` in the code and both structures have the same length, the code is valid.

To infer invariants, our algorithm starts from the strongest possible ones then iterates by relaxing them as needed until it finds a fixed point, as we illustrate in Algorithm 2. The starting point is the program states resulting from symbolically executing the network function's initialization code. At this point, the invariants are the strongest possible ones: "all maps will always be exactly as they are after initialization".

```
function FindFixedPoint(code)
    $invs = InitialInvariants(code)$
    $states = InitialStates(code)$
    do
        $invs = Relax(invs, states)$
        $states = SymbolicallyExecute(states, invs)$
    while $invs$ do not hold on $states$
    return $invs$
```

Algorithm 2: Core of the invariant inference algorithm.

The initial invariants are unlikely to hold on the packet processing code, unless the code does nothing. After symbolically executing the packet processing code for one iteration of the network function's infinite packet-processing loop, the engine *relaxes* the invariants to match the program states that result from the iteration. For instance, the initial invariant "the firewall's `flow_table` is always empty" could be relaxed into "the `flow_table` may have items, and its length is always the same as the `statistics`". The engine then symbolically executes a packet-processing iteration again using these new invariants, which may lead the engine to explore new paths in the code such as a path in which the packet's flow is found in the `flow_table`, previously infeasible as the table was assumed to be empty. The engine then relaxes the invariants again, and executes an iteration with these new invariants, until the invariants no longer need relaxing after an iteration. By definition, the final relaxation yields invariants that hold on the initial state as well, thus the result is a correct set of invariants. The algorithm is guaranteed to converge since the set of invariants can only shrink. It may converge towards the empty set of invariants because the code has no invariants or because the engine could not infer any. The goal is not to find some "ideal" set of perfect invariants, only enough invariants to be able to symbolically execute and verify the code. If some properties happen to hold but are not necessary, inference need not find them.

The key to inferring useful invariants is to use ghost maps' *known* items to form invariant candidates, and then check whether these candidates hold using the *unknown* items. Thus, instead of using low-level invariant templates such as "the value is nonzero" as in Houdini [20], the tool can find invariants based on the constraints that hold on known items. For instance, the tool does not need a definition of a "device" to solve our second motivating example.

Besides finding invariants among map items and lengths, tools can also find invariants *across* maps. Such invariants are of the form "if M_1 associates key K with value V, then M_2 has some key and possibly value related to K, V". For instance, in the first motivating example, finding the invariant "all keys of `table` are also keys in `statistics`" enables the tool to prove that the code uses `increment` correctly. Inferring such invariants requires finding functions F_K, F_V, F_P for two maps M_1, M_2 such that $((get(M_1, K) = V) \wedge F_P(K, V)) \Rightarrow (get\big(M_2, F_K(K, V)\big) = F_V(K, V))$, or alternatively find only F_K, F_P and merely infer that M_2 contains $F_K(K, V)$ without inferring the associated value. Candidates for F_* are found using the known items and confirmed using the unknown items invariant.

In summary, the representation we propose for ghost maps also enables an elegant invariant inference algorithm based on finding candidate invariants and relaxing them as necessary. Representing known items explicitly lets a verification tool find useful candidates for invariants, instead of limiting itself to special-cased low-level templates.

5 Implementation

We implement our technique in Klint, a tool which uses the `angr` [48] symbolic execution engine and the Z3 [12] solver.

Klint takes a network function binary and a Python specification as inputs and proves that the binary satisfies the specification or produces a counter-example. Klint can also be used without a specification to check that no path in the code crashes or accesses memory out of bounds.

Klint first identifies all environment interactions in the binary, which correspond to data structure operations or networking operations, using the symbols that the binary must export since it dynamically links the environment code. Then, Klint maps these operations to their contracts, provided by the developers of the data structures and of the networking environment. Data structure contracts are written in terms of ghost maps, as defined in §3, while the handful of fundamental networking operations such as transmitting a packet have manually written contracts for Klint. When the binary calls its environment, Klint symbolically executes the corresponding contract instead of the implementation, inferring types and control flow information as described in §2.2: Klint knows the types of parameters to environment interactions, and extracts control flow in the form of path constraints. This enables Klint to understand the semantics of the binary under verification in terms of its environment.

Klint symbolically executes the network function's initialization code, as defined in §2.1, then infers invariants that hold in all packet-processing paths by symbolically executing the packet-processing code until it finds a fixed point, as described in §4. Klint then checks whether all packet-processing paths satisfy the specification, which indicates whether the network function binary as a whole does.

Specifications are Python programs that use ghost maps, which Klint interprets using peer symbolic execution [6] on each state resulting from symbolic execution of the binary. The binary may abstractly manipulate more than one ghost map, since it may use multiple data structures and individual data structures may be modeled by contracts as multiple maps. Thus, when the specification refers to a map, Klint must infer which of the maps it is, using heuristics to try likely candidates first, and only fail verification if the specification is violated for all possibilities. For instance, a firewall can track outgoing flows and maintain statistics per IP address. Klint must infer that the flow table in Algorithm 1 is the abstract form of the former data structure, not the latter.

Developers must comply with good programming practices such as state separation if they want verification to succeed, and Klint enforces this during verification. If developers do not separate data structure code from the rest of the network function, Klint will encounter too many paths and fail once it has executed a configurable instruction threshold. Developers already follow even stricter practices to write BPF programs due to the strict Linux verifier [10].

Klint models heap memory with ghost maps, treating memory as just another kind of data structure and including it in invariant inference. Developers use a standard `calloc`-like interface to allocate memory. During symbolic execution, memory allocations return *symbolic pointers*, ensuring that Klint will explore all paths even if some paths depend on the value of pointers. This is not the case in a tool such as Vigor [53], which uses concrete pointers.

To ensure that developers cannot "hide" memory from Klint, all memory outside of the stack and the maps-modeled heap is read-only after initialization. This is not a limitation on sensible developers, who allocate on the heap through the environment and cleanly separate mutable state.

Our memory model is similar to Memsight [9] and KLEE's segmented memory [26], but it requires almost no effort to implement thanks to the flexibility of ghost maps.

Klint can do *full-stack* verification, verifying the entire software stack, including the network driver and most of a minimal operating system.

To verify the network driver, Klint matches hardware accesses to the *actions* they correspond to, using a hardware description of the network card written in a domain-specific language and based on publicly available data. Klint intercepts reads and writes to network card registers, which usually go through port- or memory-mapped I/O. When the network driver writes a value to a register, Klint reverse-engineers what the driver might be doing by finding all actions that could match the write, and checking which ones are feasible in the current hardware state. For instance, if the driver sets the "enable promiscuous mode" flag in the network card, Klint looks up the corresponding action and checks its precondition, which states packet reception must be disabled. If the precondition does not hold, or if no action matches the write performed by hardware, Klint aborts verification. Actions can also have postconditions describing what happens to the hardware as the result of an action, such as a self-clearing bit in a register.

The only environment operations for which Klint uses contracts instead of implementations during full-stack verification are the data structures and the memory allocator due to their complexity. We verified that they obey their contracts using machine-checked proofs.

Full-stack binary verification does impose one constraint: while the network function and its network driver can be compiled together, the environment has to be compiled separately and then linked together without link-time optimizations, ensuring the symbols corresponding to environment operations exist and can be given as an input to Klint.

Our trusted computing base is made up of Klint itself, including `angr` and Z3; the bootloader; the hardware; and the VeriFast [25] theorem prover we use to verify the memory allocator and data structures. Since VeriFast verifies source code, we trust the C compiler for the data structures and memory allocator, but this is not a fundamental limitation.

6 Evaluation

We evaluate Klint using the binaries, without debug symbols, of 6 network functions: an Ethernet bridge with a spanning tree protocol, a firewall, an implementation of Google's Maglev [16], a network address translator, a traffic policer, and an IPv4 router with longest prefix matching. The first five are based on publicly available code from the Vigor [53] project, which verified source code.

We show that Klint can quickly verify binaries in §6.1, that it reduces the trusted base and enables developers to write faster code in §6.2, and that it is applicable to real-world code including BPF in §6.3.

6.1 Verifying network functions

We summarize the time it takes Klint to verify our network functions in Table 1, split into the time to symbolically execute the code, infer invariants, and verify the resulting paths. Klint runs multiple iterations of symbolic execution and invariant inference to find a fixed point, thus we report the total time spent in each category.

We measure all times on an Intel i7-7700HQ CPU running at 3.60GHz. Klint is single-threaded, though invariant inference is embarrassingly parallelizable. We prototyped parallelization but ran into a complex bug between angr and Z3 due to garbage collection [1], and since Klint is fast enough we did not investigate further.

Overall, even the most complex of our network functions, the bridge, takes about 2 minutes to verify, which we believe is reasonable. We could further reduce verification time by reducing the redundancy in some invariants, and by using less flexible tools than angr and Z3, since we do not use their full power. The router uses a single data structure, the longest-prefix-match table, thus it only has one invariant.

We caution against over-interpreting the exact results, as most of the time is spent by the Z3 solver, and we noticed that the time Z3 takes to solve queries can vary significantly with small changes in queries, due to the heuristic-based nature of solving. Thus, total verification time can vary by dozens of seconds with minor changes in Klint or Z3.

	Time (seconds)				**#invs**
	Sym. ex.	**Inv. inf.**	**Verif.**	**Total**	
Bridge	82	18	19	**119**	32
Firewall	26	7	10	**43**	20
Maglev	36	11	10	**57**	25
NAT	43	7	8	**58**	20
Policer	53	10	6	**69**	25
Router	0	0	1	**1**	1

Table 1. Our network functions, the time Klint takes to verify them, and the number of invariants it finds.

```
1  def spec(pkt, config, sent_pkt):
2    if pkt.ipv4 is None or pkt.tcpudp is None:
3      assert sent_pkt is None
4      return
5    if pkt.device == config["external device"]:
6      flow = {
7        'src_ip': pkt.ipv4.dst,
8        'dst_ip': pkt.ipv4.src,
9        'src_port': pkt.tcpudp.dst,
10       'dst_port': pkt.tcpudp.src,
11       'protocol': pkt.ipv4.protocol
12     }
13     # there must exist a map that tracks flows,
14     # regardless of what it maps them to
15     table = Map(typeof(flow), ...)
16     if sent_pkt is not None:
17       assert flow in table.old
18       assert sent_pkt.data == pkt.data
19       assert sent_pkt.device == 1 - pkt.device
```

Listing 2. Partial specification of a firewall with 2 devices. This specification can be given to Klint unmodified.

We show the Klint equivalent of Algorithm 1, our running example of a specification, in Listing 2. The full specification is too long to show here, but Klint can also verify this partial specification.

The specification abstracts away implementation details such as the type of values in the firewall's flow table, as seen in line 15. This uses Python's "ellipsis" literal, meant for domain-specific languages.

This specification is actual Python code run in the standard Python interpreter by Klint, thus developers can write specifications using their existing programming knowledge. Klint currently requires the order of fields in specification structures such as the flow declared in line 6 to match the order in the corresponding implementation structure. Having Klint try all possible orders would not finish in reasonable time, but there may be better strategies.

We found and fixed two implementation bugs using specifications we wrote based on standards. First, section 7.8 of the IEEE 802.1D standard [30] forbids adding Ethernet group addresses to the filtering table, which we originally forgot to implement. Second, our NAT originally misused its port allocator if configured to use only a sub-range of ports. This was already present in the Vigor NAT, because Vigor requires a concrete size for data structures during verification, and its authors only verified it for the full port range.

Verifying the entire stack requires an additional 15 minutes per network function, most of which is spent inferring the network driver's actions because the driver performs thousands of writes to registers and each of these writes requires work from Klint. This could be improved by recognizing simple loops, since most initialization operations are performed in loops over categories of registers.

6.2 Faster verified network functions

Klint enables developers to write faster verified network functions in two ways: developers can quickly prototype changes to data structures without having to verify them first, and they can use the abstractions they want instead of verification-specific ones. In practice, we expect developers to either use existing verified data structures or write contracts for existing "trusted" data structures such as BPF maps. Klint enables this workflow by only requiring contracts and not proofs for these trusted data structures, whereas previous tools required an all-or-nothing approach.

First, ghost maps and invariant inference enable quick prototyping of new data structures. For instance, we rewrote our port allocator to have different performance characteristics with slightly different semantics, including stricter preconditions as we believed our network functions did not need the generality of the existing ones. Using an approach such as Vigor or Gravel, even when prototyping, we would have needed to add a model of the new port allocator to the verification tool, including annotations for invariants, to check whether our network functions would still be correct when using it. With Klint, we wrote new contracts, automatically verified that our network functions already satisfied the new stricter preconditions, then manually verified the implementation once we finished prototyping.

Second, verifying binaries enables simpler and faster code by removing abstractions over low-level hardware features that existed for the sake of verification. For instance, using a tool such as Vigor, obtaining the time from the environment requires calling a C function that the tool replaces with a model at verification time. Unless the compiler can inline this function, the developer will pay the performance cost at run time. With Klint, the code can use inline assembly to read the CPU's time stamp counter, which Klint handles in the same way as other assembly instructions. Furthermore, in the Vigor model, the C function must be verified separately using a different tool, a problem Klint does not have.

	Vigor			Klint		
	Tput	Latency (*us*)		Tput	Latency (*us*)	
	(*Gb/s*)	50%	99%	(*Gb/s*)	50%	99%
Bridge	5.54	3.98	4.25	10*	3.84	4.26
Firewall	7.77	3.92	4.26	10*	3.84	4.25
Maglev	6.34	3.96	4.28	10*	3.90	4.27
NAT	3.47	3.97	4.32	10*	3.87	4.27
Policer	9.12	3.87	4.25	10*	3.83	4.24

Table 2. Maximal single-link throughput without loss, and latency with 1 Gb/s background load of the original Vigor network functions and our versions. * = link saturated

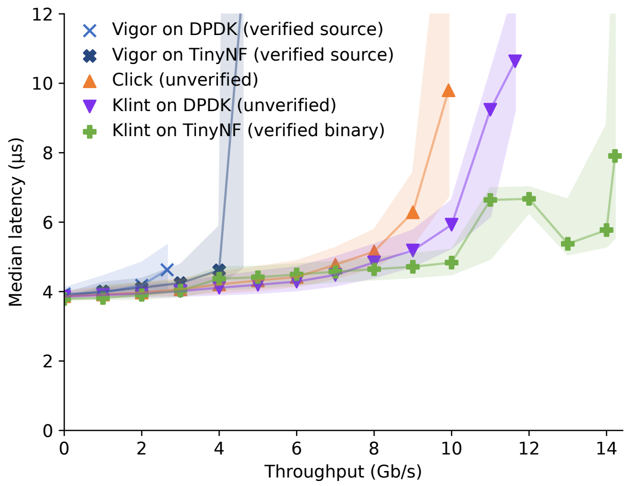


Figure 5. Throughput without loss vs. median latency of different bridges. Shaded areas delimit 5th and 95th percentiles.

We benchmark our network functions using the Vigor ones as baselines, with the TinyNF [45] driver instead of the original DPDK subset since it is faster and is the base for our own network driver. We use the same setup as Vigor to make the comparison useful: two machines as in RFC 2544 [4], one running a network function and one running the MoonGen packet generator [17]. Both machines have Intel Xeon E5-2667 v2 CPUs at 3.30GHz, Intel 82599ES NICs, and run Ubuntu 18.04. These network cards are the ones we modeled for Klint's full-stack verification. We measure throughput with minimally-sized packets, filling network functions' flow tables to 90% of their capacity.

We first run the same benchmark used in the Vigor paper: find the maximum throughput the network functions can sustain without dropping packets using one 10 Gb/s link and measure their latency when fed 1 Gb/s of background load. All our network functions can handle 10 Gb/s whereas the Vigor ones cannot, as we show in Table 2.

Since the Ethernet link is a bottleneck in the Vigor benchmarks, we use two links, for a theoretical maximal throughput of 20 Gb/s. To obtain more details about performance, we measure latency at increments of 1 Gb/s until the network function drops packets. We focus on the bridge for lack of space. We included the original Vigor bridge running on its verified DPDK subset, the Vigor bridge running on the verified TinyNF driver, the Click [28] bridge originally used as a baseline by Vigor, our bridge running on our verified driver, and our bridge running on DPDK, which is not verified. We do not know of any "standard" network functions we could use as a baseline beyond Click.

Despite our bridge having extra features compared to the Vigor one, such as support for a spanning tree protocol, it can reach more throughput before dropping packets than any of the other bridges, including the bridge from the widely used Click toolchain, as we show in Figure 5.

	Language	Data structures	Extra inputs	Pointer arith.	Unbounded loops	Precise
Gravel [55]	LLVM	C++ STL	Intermediate specs	Limited	No	Yes
Prevail [21]	BPF	BPF maps	No need	If safe	Yes	No
Vigor [53]	LLVM	Custom "libVig"	NF-specific models	No	No	Yes
Klint	x86_64	Any, w/ contracts	No need	If safe	No	Yes

Table 3. Comparison of this work with previous network function verification efforts.

6.3 Applicability

Klint is applicable to real-world network functions: it does not require additional inputs from developers, it enables operators to verify the entire stack of network functions, its network function model is a superset of the widely used BPF, and it enables developers to use any programming language including those not considered mature enough to be trusted.

As we show in Table 3, Klint operates on binaries, can handle any data structure for which map-based contracts are provided, does not require intermediate inputs, does not limit developers' use of pointer arithmetic beyond memory safety, and precisely tracks the contents of data structures and packets enabling functional correctness proofs.

Previous work falls into two categories. Vigor and Gravel prove functional correctness but require a typed intermediate language, extra inputs, and specific data structures. Prevail [21] handles BPF bytecode and maps, which BPF developers must use anyway, and can even handle unbounded loops, but can only prove memory safety and crash freedom. Prevail can be viewed as a superset of the Linux BPF verifier. Klint provides the best of both worlds: it requires neither a typed intermediate representation nor extra inputs, and it does not limit data structures, while also enabling proofs of functional correctness. As an example of unnecessary inputs, we were able to remove around 3000 lines of proofs for invariants when porting Vigor network functions to Klint.

Operators can verify the entire software stack, and thus do not need to trust software such as network drivers, using Klint's full-stack verification. Developers can thus tune the drivers for performance by removing features they do not need, or even rewrite their own driver to suit it to a specific usage pattern. PacketMill [19] showed that such transformations can be done with developer hints to increase network function performance.

Klint can be used to verify network functions in containers, i.e., statically linked with a Linux implementation of the environment abstraction and running within Docker [34]. Containers are a convenient way to deploy and manage programs and have been proposed as a deployment model for network functions [50]. Klint can verify such network functions in the same way it verifies full-stack ones, although the trusted code base is larger since operators need to trust the container runtime as well as the Linux environment implementation running inside the container.

Our network function model is a superset of BPF. The Linux kernel verifies that BPF programs are memory safe and have no unbounded loops. Klint verifies functional correctness, though it also requires a lack of unbounded loops. BPF requires developers to use a fixed set of data structures, mostly maps. Klint enables developers to use any data structure that has contracts based on ghost maps.

To show that Klint's model is at least as expressive as BPF's, we use five existing BPF programs: Facebook's Katran [47] load balancer, the CRAB [27] load balancer, a filter from Suricata [44], a firewall from hXDP [5], and a bridge from Polycube [35]. We extend Klint to analyze the assembly code compiled by the kernel after dumping it through a Linux debugging facility, using contracts we wrote for the BPF maps. Since we have no specifications for these BPF programs, Klint only verifies memory safety and crash freedom. Klint verifies the bridge and firewall in seconds, and CRAB and the Suricata filter in less than 2 minutes, but Katran requires almost 4 hours. This is due to our choice to model packet contents with a ghost map for simplicity, even though packet contents do not factor into invariant inference. Katran reads and writes to dozens of fields in the packet, which means the ghost map representing the packet has too many items to be efficient. This could be fixed by using a different way to model packet memory, such as Z3 arrays.

Developers can use any programming language, even if that language is considered too "exotic" or "immature" for operators to blindly trust it, since Klint verifies binaries, and thus can catch errors resulting from compiler bugs.

For instance, the Rust [39] programming language is a promising direction for writing low-level systems code, including networking, but a developer or an operator might be concerned about its maturity level. Indeed, as of this writing there are currently 69 issues in the Rust bug tracker [46] marked with the "unsound" tag, meaning the compiler allows code that violates Rust's safety guarantees. Furthermore, some Rust features such as removing unused parts of the standard library are currently experimental.

However, we are able to write a traffic policer in Rust, compile it using these experimental features, and verify it using Klint with the same specification as our C implementation. We can thus be confident that no matter what bugs lie in the Rust compiler, our binary is correct.

7 Limitations

KLINT's main limitation is that it cannot handle multi-threaded network functions that share state across threads. As we stated in §2, parallel code causes path explosion due to the amount of possible interleavings among threads. Running isolated instances of a network function in parallel can work by steering flows to cores, in which case KLINT can verify the instances, but this has skew issues.

KLINT imposes the following performance limitation on network functions: data structures must be dynamically linked so that KLINT can identify their operations by mapping the symbols left undefined in the binary to the contracts it is given as inputs. This can lead to slower function calls and lost opportunities for inlining.

KLINT also has the non-fundamental limitation of only supporting packet arrival as a trigger to run code and not timers or other events. Extending KLINT to support this is engineering work, replacing the restricted event handler model of packet reception with a more general one that branches on the event type.

Verifying network functions is only one part of the puzzle. KLINT can verify that a network function implements a protocol, but not that the protocol itself satisfies properties such as avoiding infinite message loops. Specialized techniques exist to verify protocols, such as IronFleet [22].

The remaining part of the puzzle is data structure implementations, for which manual verification tools currently remain necessary. We used VeriFast [25], which uses annotation for C, but other approaches exist such as Dafny [31], a programming language designed for verification.

8 Related work

Runtime verification is a related but distinct area of work, focusing on checking behavior at runtime. This catches bugs even in code currently beyond the reach of formal verification, for instance due to complex parallelism. Tools such as Aragog [52] trade completeness and performance for applicability. They only look at input and output packets and view network functions as "black boxes", thus they impose no constraints on code. But they cannot guarantee the absence of bugs, and in distributed environments cannot prevent bugs that can only be detected after compiling information from different machines. They also impose runtime overheads, unlike KLINT, due to the runtime nature of checks.

Symbolic execution is the main technique we build upon, and ghost maps may be useful beyond network functions. Symbolic execution is often used for bug finding instead of verification because of path explosion, but it can be augmented with techniques to bypass path explosion. Indeed, KLEE [7] and `angr` [48] were both designed primarily to find bugs, yet Vigor [53] and KLINT show that they can be used for verification. S2E [8] was also designed to find bugs but reused for network function verification by Dobrescu and Argyraki [14]. Serval [40] is a symbolic execution engine enhanced with verification techniques and can prove systems such as a security monitor, at the cost of requiring some human annotations. KLINT could have used Serval as a base; we chose `angr` mostly because it is designed for quick prototyping. Path explosion can also be bypassed by writing code with few paths, as in the Hyperkernel [42].

Using maps as part of analyzing programs has been proposed before, though previous approaches were not aimed at functional verification, such as the Memsight [9] memory model for symbolic execution, or the technique proposed by Dillig et al. [13] to verify memory safety.

Network function verification tools such as Vigor [53], Gravel [55], and Prevail [21], which require source code, all inspired our design. Bolt [24] and Pix [23] verify performance instead of correctness, and require source code, though we believe they could use our techniques to only require binaries. While verifying binaries provides key advantages, verifying source code makes debugging failed verification easier since compiler optimizations can make it hard to match what the binary does wrong to what the code does wrong.

BPF is a more applicable but weaker form of network function verification: an in-kernel "verifier" checks memory safety and crash freedom, allowing untrusted code to run in kernel mode for performance without safety risks. BPF verifiers are fast, at the cost of restricting the code developers may write. BPF code cannot contain unbounded loops, must use specific data structures, and must include explicit checks for out-of-bounds memory accesses, even if a "smarter" and slower verifier might not need these checks. Internet infrastructure companies such as Cloudflare [32] use BPF as a core part of their infrastructure. Prevail [21] showed that formal methods can help BPF verification scale. Verified interpreters [51] and just-in-time compilers [41] for BPF exist, but they make no promises about functional correctness.

9 Conclusion

We presented KLINT, an automated tool to formally verify that network function binaries satisfy specifications with neither source code nor debugging symbols, given contracts for trusted data structures. This enables developers to provide guarantees about proprietary code to operators, removing a key barrier for adoption of formally verified network functions. KLINT uses maps as "universal" data structures for both specifications and contracts, enabling a sweeping simplification in reasoning including invariant inference. Using KLINT, we verified the functional correctness of 6 network function binaries written in C and 1 in Rust, and the memory safety and crash freedom of 5 existing BPF programs.

Acknowledgments

We thank the anonymous reviewers and our shepherd Jay Lorch for improving this paper, and `angr` maintainer Audrey Dutcher for her quick fixes to the few issues we encountered.

References

[1] Angr contributors. angr issue 938: SEGFAULT libz3. https://github.com/angr/angr/issues/938.

[2] BPF authors and contributors. bpf-helpers(7) Linux manual page. https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[3] Bradley, A. R., Manna, Z., and Sipma, H. B. What's decidable about arrays? In *Intl. Conf. on Verification, Model Checking, and Abstract Interpretation* (2006).

[4] Bradner, S., and McQuaid, J. Benchmarking methodology for network interconnect devices. RFC 2544, RFC Editor, 1999.

[5] Brunella, M. S., Belocchi, G., Bonola, M., Pontarelli, S., Siracusano, G., Bianchi, G., Cammarano, A., Palumbo, A., Petrucci, L., and Bifulco, R. hxdp: Efficient software packet processing on FPGA NICs. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[6] Bruni, A., Disney, T., and Flanagan, C. A peer architecture for lightweight symbolic execution. http://hoheinzollern.files.wordpress.com/2008/04/seer1.pdf, Unpublished.

[7] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2008).

[8] Chipounov, V., Georgescu, V., Zamfir, C., and Candea, G. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems (HOTDEP)* (2009).

[9] Coppa, E., D'Elia, D. C., and Demetrescu, C. Rethinking pointer reasoning in symbolic execution. In *ACM Intl. Conf. on Automated Software Engineering (ASE)* (2017).

[10] Corbet, J. Bounded loops in BPF programs. https://lwn.net/Articles/773605/.

[11] Corbet, J. The BPF system call API, version 14. https://lwn.net/Articles/612878/.

[12] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2008).

[13] Dillig, I., Dillig, T., and Aiken, A. Precise reasoning for programs using containers. In *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)* (2011).

[14] Dobrescu, M., and Argyraki, K. Software dataplane verification. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2014).

[15] DPDK: Data plane development kit. https://dpdk.org.

[16] Eisenbud, D. E., Yi, C., Contavalli, C., Smith, C., Kononov, R., Mann-Hielscher, E., Cilingiroglu, A., Cheyney, B., Shang, W., and Hosein, J. D. Maglev: A fast and reliable software network load balancer. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2016).

[17] Emmerich, P., Gallenmüller, S., Raumer, D., Wohlfart, F., and Carle, G. MoonGen: A scriptable high-speed packet generator. In *Internet Measurement Conf. (IMC)* (2015).

[18] Equinix. Network edge | Equinix edge services. https://www.equinix.se/services/edge-services/network-edge.

[19] Farshin, A., Barbette, T., Roozbeh, A., Maguire Jr., G. Q., and Kostić, D. PacketMill: Toward per-core 100-Gbps networking. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2021).

[20] Flanagan, C., and Leino, K. R. M. Houdini, an annotation assistant for ESC/Java. In *Intl. Symp. on Formal Methods Europe* (2001).

[21] Gershuni, E., Amit, N., Gurfinkel, A., Narodytska, N., Navas, J. A., Rinetzky, N., Ryzhyk, L., and Sagiv, M. Simple and precise static analysis of untrusted Linux kernel extensions. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (2019).

[22] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J., Parno, B., Roberts, M. L., Setty, S., and Zill, B. IronFleet: Proving practical distributed systems correct. In *ACM Symp. on Operating Systems Principles (SOSP)* (October 2015), ACM.

[23] Iyer, R., Argyraki, K., and Candea, G. Performance interfaces for network functions. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2022).

[24] Iyer, R., Pedrosa, L., Zaostrovnykh, A., Pirelli, S., Argyraki, K., and Candea, G. Performance contracts for software network functions. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2019).

[25] Jacobs, B., and Piessens, F. The VeriFast program verifier, 2008.

[26] Kapus, T., and Cadar, C. A segmented memory model for symbolic execution. In *ACM SIGSOFT Intl. Symp. on the Foundations of Software Engineering (FSE)* (2019).

[27] Kogias, M., Iyer, R., and Bugnion, E. Bypassing the load balancer without regrets. In *Symp. on Cloud Computing (SOCC)* (2020).

[28] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click modular router. *ACM Transactions on Computer Systems (TOCS) 18*, 3 (2000).

[29] Kuznetsov, V., Kinder, J., Bucur, S., and Candea, G. Efficient state merging in symbolic execution. In *Intl. Conf. on Programming Language Design and Implementation (PLDI)* (2012).

[30] LAN/MAN Standards Committee. IEEE standard for local and metropolitan area networks: Media access control (MAC) bridges. Tech. rep., IEEE Standards Association, 2014. IEEE Std 802.1D-2004.

[31] Leino, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Intl. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR)* (2010).

[32] Majkowski, M. Cloudflare architecture and how BPF eats the world. https://blog.cloudflare.com/cloudflare-architecture-and-how-bpf-eats-the-world/.

[33] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference* (San Diego, CA, Jan. 1993), USENIX Association.

[34] Merkel, D. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal* (2014).

[35] Miano, S., Bertrone, M., Risso, F., Bernal, M. V., Lu, Y., Pi, J., and Shaikh, A. A service-agnostic software framework for fast and efficient in-kernel network services. In *ACM/IEEE Symp. on Architectures for Networking and Communications Systems* (2019).

[36] MITRE Corporation. MS13-064. Available from CVE Details, CVE-ID MS13-064., 2013.

[37] MITRE Corporation. CVE-2014-9715. Available from CVE Details, CVE-ID CVE-2014-9715., 2014.

[38] MITRE Corporation. CVE-2015-6271. Available from CVE Details, CVE-ID CVE-2015-6271., 2015.

[39] Mozilla Research. Rust programming language. https://www.rust-lang.org/.

[40] Nelson, L., Bornholt, J., Gu, R., Baumann, A., Torlak, E., and Wang, X. Scaling symbolic evaluation for automated verification of systems code with Serval. In *ACM Symp. on Operating Systems Principles (SOSP)* (2019).

[41] Nelson, L., Geffen, J. V., Torlak, E., and Wang, X. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[42] Nelson, L., Sigurbjarnarson, H., Zhang, K., Johnson, D., Bornholt, J., Torlak, E., and Wang, X. Hyperkernel: Push-button verification of an OS kernel. In *ACM Symp. on Operating Systems Principles (SOSP)* (2017).

[43] Network Working Group. RFC 1812, requirements for IP version 4 routers. https://www.rfc-editor.org/rfc/rfc1812.txt, 1995.

[44] Open Information Security Foundation. Suricata website. https://suricata.io/.

[45] Pirelli, S., and Candea, G. A simpler and faster NIC driver model for network functions. In *Symp. on Operating Systems Design and*

Implementation (OSDI) (2020).

[46] Rust authors and collaborators. Issues - rust-lang/rust. https://github.com/rust-lang/rust/issues.

[47] Shirokov, N., and Dasineni, R. Open-sourcing Katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer, May 2018.

[48] Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., and Vigna, G. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symp. on Security and Privacy (S&P)* (2016).

[49] Turing, A. M. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* (01 1937).

[50] Wang, J., Lévai, T., Li, Z., Vieira, M. A. M., Govindan, R., and Raghavan, B. Galleon: Reshaping the square peg of NFV, 2021.

[51] Wang, X., Lazar, D., Zeldovich, N., Chlipala, A., and Tatlock, Z. Jitk: A trustworthy in-kernel interpreter infrastructure. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2014).

[52] Yaseen, N., Arzani, B., Beckett, R., Ciraci, S., and Liu, V. Aragog: Scalable runtime verification of shardable networked systems. In *Symp. on Operating Systems Design and Implementation (OSDI)* (2020).

[53] 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 *ACM Symp. on Operating Systems Principles (SOSP)* (2019).

[54] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K., and Candea, G. A formally verified NAT. In *ACM SIGCOMM Conf. (SIGCOMM)* (2017).

[55] Zhang, K., Zhuo, D., Akella, A., Krishnamurthy, A., and Wang, X. Automated verification of customizable middlebox properties with Gravel. In *Symp. on Networked Systems Design and Implementation (NSDI)* (2020).

Appendix A Verified properties

We used Klint to verify both network functions we wrote and existing BPF network functions. We summarize the properties we proved for our network functions in this appendix. The BPF network functions have no formal specification and writing one would require a discussion with their authors to understand which edge cases are intended and which are not, so we instead chose to only verify that they are memory safe and free of crashes.

The full specifications are available on our repository: https://github.com/dslab-epfl/klint.

Bridge: we wrote a specification by manually extracting properties from the IEEE 802.1D [30] standard. For instance, section 7.7.1 of the standard, "Active topology enforcement", states that "Each Port is selected as a potential transmission Port if, and only if [...] The Port considered for transmission is not the Port on which the frame was received [...]", thus our specification checks that if the packet was transmitted, the transmission port must not be the reception port. As we explain in Section 6.1, we found a bug after translating section 7.8 of the standard, "The Learning Process", which states a condition for learning an address in the filtering database: "the source address field of the frame denotes a specific end station (i.e., is not a group address)".

Klint helps us write this specification by allowing us to write properties that must hold on packets and on state without having to explicitly depend on the bridge's internals. For instance, we do not need to specify the data type that the bridge uses to store metadata about Ethernet addresses, only that the bridge conceptually has a map with Ethernet addresses as keys, and that the source address of an incoming packet is or is not added depending on specification-related factors.

Firewall: beyond the partial specification of Listing 2, we wrote a full specification that is an evolution of the one from Vigor [53]. We model the firewall's state as a map from flows to last update time and ensure that the firewall (1) adds flows from the internal network to the state if possible, refreshing their last update time if necessary, and (2) only lets packets from the external network through if their flow belongs to the state. The specification also ensures that the firewall does not modify packet contents and does not drop packets unless they are incoming packets with no matching flow in the state.

Our specification is not concerned with how the firewall "remembers" flows, nor with when exactly this happens in packet processing, only that outgoing packets flows must be remembered if there is space and that incoming packets must be of a known flow if they are forwarded.

As we show in §6.1, Klint can also verify a more restricted specification that is only concerned with what happens when a packet is transmitted, including the fact that its flow must have been known previously.

Maglev: while we do not know of a formal specification for Maglev, our specification is an evolution of the one from Vigor [53], which was written according to the behavior described by the Google paper [16]. We model the state as two maps, one from flows to backends and one from backends to last heartbeat time. Packets from backends are heartbeats and must update the corresponding last heartbeat time and then be dropped. Packets from clients must be routed to a backend and must not be dropped unless there are no available backends.

NAT: our specification is an evolution of the one from VigNAT [54], which fully describes the behavior of the NAT. We model the state as a map from flows to last update time, in a similar fashion to the firewall specification. Packets must only be dropped if they are not IP or TCP/UDP, as the NAT does not support other protocols (for now), or if they come from the external network but do not belong to a known flow. Packet headers must be updated according to the state, and packets from the internal network must trigger state updates.

One interesting aspect of Klint with regards to the NAT is invariant inference: Vigor's original NAT required about 3000 lines of manually written proof annotations for the invariants between the three data structures the NAT uses internally, and the Vigor specification was dependent on these data structures. Klint instead infers these invariants automatically, and our specification defines the NAT in terms of maps.

Policer: our specification is an evolution of the one from Vigor [53], which fully describes the behavior of the policer. We model the state as one map from buckets to tokens, and one map from IP addresses to buckets. The specification then enforces that the policer must update the state according to incoming packets and their size and drop packets if their bucket is too full.

Router: we wrote a specification based on RFC 1812 [43], "Requirements for IP Version 4 Routers". We model the state as a map from CIDR blocks to devices. We enforce the header validation requirements from section 5.2.2 of the RFC, the time to live requirements from section 4.9.9.2, and most importantly the longest-prefix-match property to find the next hop address from section 5.2.4.3.

Appendix B Ghost maps *get* algorithm

We present here the algorithm for the *get* operation on ghost maps. *PC* is the Path Constraint. UK_M, UV_M, and UP_M are the triple forming map M's unknown item, as we explain in §3.3. The *condition* and *value_hint* are those required to handle invariant recursion, as we explain in §3.4.

```
function KnownSize(M)
    $result = 0, known = \emptyset$
    for $(k, v, p)$ in M's known items do
        $result$ += $ITE(k \notin known \land p, 1, 0)$
        $known$ += $k$
    return $result$

function Get(M, K)
    for $(k, v, p)$ in M's known items do
        if $unsatisfiable(condition \land K \neq k)$ then
            return $(v, p)$
    if $unsatisfiable(condition \land K \neq UK_M)$ then
        return $(UV_M, UP_M)$
    Let $(V, P)$ be a fresh value and presence bit
    if condition is set then
        Add $condition \Rightarrow V = value\_hint$ to the PC
    Let $U = \bigwedge(K \neq K')$ for each known key $K'$ in $M$
    Add $(K, V, P)$ to $M$'s known items
    for $(k, v, p)$ in M's items do
        Add $K = k \Rightarrow \big((V = v) \land (P = p)\big)$ to the PC
    Add $U \Rightarrow invariant_M(K, V, P)$ to the PC
    Add $KnownSize(M) \leq length(M)$ to the PC
    return $(V, P)$
```