
# HILTI: An Abstract Execution Environment for Concurrent, Stateful Network Traffic Analysis

Robin Sommer,[†*] Lorenzo De Carli,[§] Nupur Kothari,[†]
Matthias Vallentin,[‡] and Vern Paxson[†‡]

TR-12-003

January 2012

## Abstract

When building applications that process large volumes of network traffic – such as firewalls, routers, or intrusion detection systems – one faces a striking gap between the ease with which the desired analysis can often be described in high-level terms, and the tremendous amount of low-level implementation details one must still grapple with for coming to an efficient and robust system. We present a novel environment, HILTI, that provides a bridge between these two levels by offering to the application designer the abstractions required for effectively describing typical network analysis tasks, while still being designed to provide the performance necessary for monitoring Gbps networks in operational settings. The new HILTI middle-layer consists of two main pieces: an abstract machine model that is specifically tailored to the networking domain and directly supports the field's common abstractions and idioms in its instruction set; and a compilation strategy for turning programs written for the abstract machine into optimized, natively executable task-parallel code for a given target platform. We have developed a prototype of the HILTI environment that fully supports all of the abstract machine's functionality, and we have ported a number of typical networking applications to the new environment. We also discuss how HILTI's processing can transparently integrate custom hardware elements where available as well as leverage non-standard many-core platforms for parallelization.

† ICSI, 1947 Center St., Ste. 600, Berkeley, CA 94704
* Lawrence Berkeley National Laboratory, 1 Cyclotron Rd., Berkeley, CA 94720
§ University of Wisconsin-Madison, 702 West Johnson Street #1101, Madison, WI, 53715
‡ UC Berkeley, Berkeley, CA 94720

This work was partially supported by funding provided to ICSI through National Science Foundation grant CNS-0915667 (“A High-Performance Abstract Machine for Network Intrusion Detection”), and by Cisco Research Award “A Concurrency Model for Deep Stateful Network Security Monitoring.” Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators and do not necessarily reflect the views of the National Science Foundation or Cisco Systems.

# 1 Introduction

Deep, stateful network packet analysis is a crucial building block for applications processing network traffic. However, when building systems such as firewalls, routers, and network intrusion detection systems (NIDS), one faces a striking gap between the ease with which one can often describe the desired analysis in high-level terms ("search for this pattern in HTTP requests"), and the tremendous amount of low-level implementation details that one must still grapple with to realize an efficient and robust implementation. When applications reconstruct the network's high-level picture from zillions of packets, they must not only operate efficiently to achieve line-rate performance under real-time constraints, but also deal securely with a stream of untrusted input that requires fail-safe, conservative processing.

Despite such challenges, however, our community sees little reuse of existing, well-proven functionality across networking applications. While individual efforts invest significant resources into optimizing their particular implementations, new projects cannot readily leverage the experiences that such systems have garnered through years of deployment, and thus have to build much of same functionality from scratch each time.

In this work we present a novel *platform* for building network traffic analysis applications, providing much of the standard low-level functionality without tying it to a specific analysis structure. Our system consists of two parts: *(i)* an *abstract machine model* specifically tailored to the networking domain and directly supporting the field's common abstractions and idioms in its instruction set; and *(ii)* a compilation strategy for turning programs written for the abstract machine into optimized, natively executable code. At the core of the abstract machine model is a *high-level intermediary language for traffic inspection (HILTI)* that provides high-level data structures, powerful control flow primitives, extensive concurrency support, and a secure memory model protecting against unintended control and data flows.

Conceptually, HILTI provides a *middle-layer* sitting between the operating system and a *host application*. HILTI operates invisibly to an application's end-users, and specifically they do not interact with it directly. Instead, an application leverages HILTI by *compiling* the analysis it wants to perform from its own custom high-level description (like a firewall's rules, or an NIDS's signature set) into HILTI code; and HILTI's compiler then translates it further down into natively executable code.

We have developed a prototype of the HILTI compiler that fully supports all of the design's functionality, and we have ported several sample networking applications to the HILTI machine model to demonstrate the aptness of its abstractions, including the BinPAC protocol parser generator [40] and the Bro NIDS's script interpreter [41]. HILTI targets production usage, with performance suitable for real-time analysis of high-volume network traffic in large-scale operational environments. While our prototype implementation cannot yet provide such performance, we show that its bottlenecks are implementation deficits that we can overcome with further engineering effort to eventually build a production-quality toolchain.

One of HILTI's main objectives is to integrate much of the knowledge that the networking community has collected over many years into a single platform for applications to build upon. As such, we envision HILTI to eventually become a framework that ships with an extensive library of reusable higher-level components, such as packet reassemblers, session tables with built-in state management, and parsers for specific protocols. By providing both the means to implement such components as well as the glue for their integration, HILTI can allow application developers to focus on their core functionality, relieving them from low-level technical challenges that others have already solved numerous times before.

We structure the remainder of this paper as follows. In §2 we motivate our work by examining the potential for sharing functionality across networking applications. We present HILTI's design in §3 and showcase four example applications in §4. Thereafter, we discuss our prototype implementation in §5 and evaluate its design in §6. We highlight specific design choices in §7 and summarize related work in §8. Finally, we conclude in §9.

# 2 The Potential for Sharing Functionality

Internally, different types of networking applications—such as packet filters, stateful firewalls, routers, switches, intrusion detection systems, network-level proxies, and even OS-level packet processing—all exhibit a similar structure built around a common set of domain-specific idioms and components.[1] While implementing such standard functionality is not rocket science, coming to a robust and efficient system is nevertheless notoriously difficult. In contrast to other domains where communities have developed a trove of standard tools and libraries (e.g., in HPC, cryptography), we find little reuse of well-proven functionality across networking systems even in the open-source world. We examined the code of three open-source networking applications of different types: iptables (firewall), Snort (NIDS), and XORP (software router). All three implement their own versions of

[1] To simplify terminology, throughout our discussion we use the term "networking application" to refer to a system that processes network packets directly in wire format. We generally do not consider other applications that use higher-level interfaces, such as Unix sockets. While these can benefit from HILTI as well, they tend to have different characteristics that would exceed the scope of our discussion.

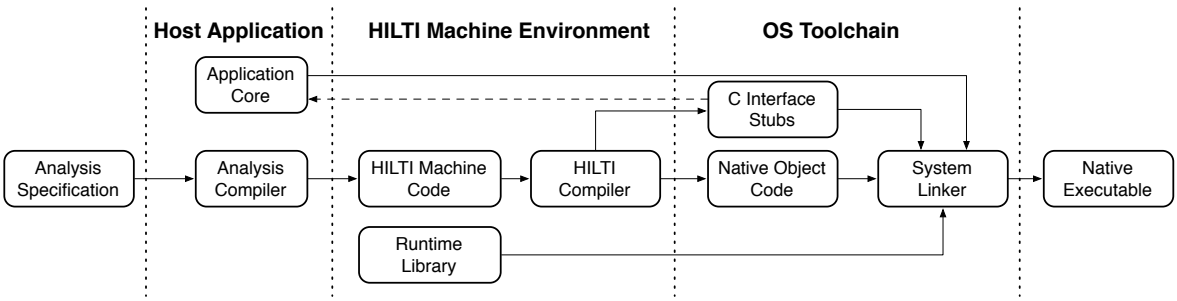



Figure 1: Workflow for using HILTI.

standard data structures with state management, support for asynchronous execution, logic for discerning IPv4 and IPv6 addresses, and protocol inspection. We also compared the source code of three well-known open-source NIDS implementations (Bro, Snort, and Suricata). We found neither any significant code shared across these systems, nor leveraging of much third-party functionality. `libpcap` is the only external library to which they all link. In addition, Snort and Suricata both leverage `PCRE` for regular expression matching, while Bro implements its own engine. In a recent panel discussion, leads of all three NIDS projects indeed acknowledged the lack of code reuse, attributing it to low-level issues like incompatible program structures and data flows. HILTI aims to overcome this unfortunate situation.

More generally, from our source code analysis we identified a number of building blocks for networking applications, as illustrated in Figure 2:

**Domain-specific Data Types.** Networking applications use a set of domain-specific data types for expressing their analysis, such as IP addresses, transport-layer ports, sub-networks, and time. HILTI's abstract machine model provides these as first-class types.

**State Management.** Most applications require long-lived state as they correlate information across packet and session boundaries. HILTI provides container types with built-in state management, and timers to schedule processing asynchronously into the future.

**Concurrent Analysis.** High-volume network traffic exhibits an enormous degree of inherent parallelism [42]. Applications need to multiplex their analyses across potentially tens of thousands of data flows, and they can also exploit the parallelism for scaling across CPUs on multi-core platforms (which is however still uncommon to find in implementations, as data dependencies pose significant challenges for stateful traffic analysis). HILTI supports both forms of parallelism via incremental processing and a domain-specific concurrency model.

**Real-time Performance.** With 10 Gbps links standard even in medium-sized environments, applications deal with enormous packet volumes in real-time. HILTI not only parallelizes processing effectively, but also compiles analyses into native executables with potential for extensive domain-specific code optimization.

**Robust/Secure Execution.** Networking applications often must process untrusted input: attackers may attempt to mislead the system; and, more mundanely, real-world traffic contains much "crud" [41] not conforming to any RFC. While writing robust C code is notoriously difficult, HILTI's abstract machine model provides a contained, well-defined environment.

**High-level Standard Components.** HILTI facilitates reuse of higher-level functionality across applications by providing *(i)* a *lingua franca* for expressing their internals, and *(ii)* interfaces for integration and customization.

## 3 The HILTI Abstract Machine Model

At the heart of HILTI lies an abstract machine model tailored to network traffic analysis, consisting of an instruction set with corresponding runtime semantics and an accompanying runtime library. The HILTI compiler turns code written for the abstract machine into native executables. Our primary design goals for HILTI are: *(i)* functionality that supports a range of applications; *(ii)* abstractions that enable transparent parallelization, optimization, and acceleration; *(iii)* a comprehensive API for customization and extension; and *(iv)* performance eventually suitable for operational deployment. In the following we discuss HILTI's design in more detail. For convenience we use the name HILTI for both the abstract machine model itself and the framework that implements it, i.e., the compiler toolchain and runtime library.

### 3.1 Workflow

Figure 1 shows the overall workflow when working with HILTI. A *host application* leverages HILTI for providing its functionality. Typically, the host application has a user-supplied *analysis specification* that it wants to deploy; e.g., the set of filtering rules for a firewall, or the set

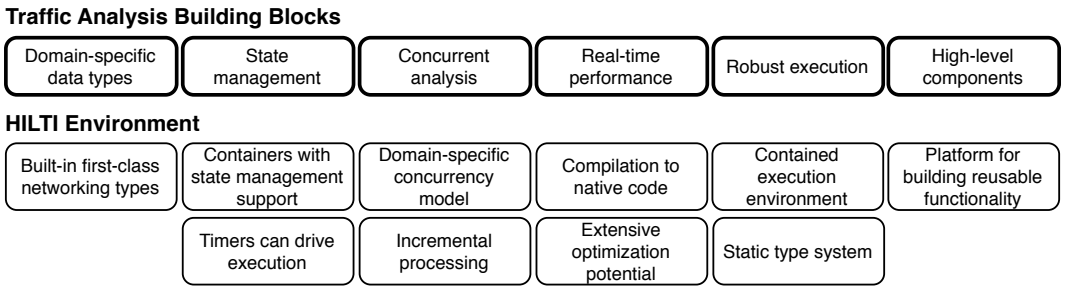



Figure 2: Building blocks of network traffic applications and how HILTI supports them.

```
# cat hello.hlt
module Main

import Hilti

void run() { # Default entry point for execution.
    call Hilti::print("Hello, World!")
}
# hilti-build hello.hlt -o a.out && ./a.out
Hello, World!
```

Figure 3: Building a simple HILTI program.

of signatures for a NIDS. The application needs to implement a custom *analysis compiler* that translates its specifications into HILTI code, which the *HILTI compiler* then turns into *object code* native to the target platform. The compiler also generates a set of *C stubs* for the host application to use to interface with the resulting code. Finally, the *system linker* combines compiled code, stubs, and the host application into a single program (either statically or JIT). Our prototype includes two tools, `hiltic` and `hilti-build`, which compile HILTI code into objects and executables, respectively (see Figure 3).

Generally, there are two ways to structure a host application. First, the HILTI code can be the main entry point to the execution, with the application providing additional functionality via external functions called out to as necessary (per Figure 3). Alternatively, the application can drive execution itself and leverage HILTI-based functionality on demand (e.g., a NIDS could feed payload into a HILTI-based protocol parser; see §4).

## 3.2 Instruction Set

**Syntax.** To keep the syntax simple, we model HILTI's instruction set after register-based assembler languages. A program consists of a series of instructions of the general form `<target> = <mnemonic> <op1> <op2> <op3>`, with target/operands omitted where not needed. In addition, there are primitives to define functions, custom data types, and local and thread-local variables.[2] By convention, mnemonics are of the form `<prefix>.<operation>`, with the same `prefix` indicating a related set of functionality. For data types in particular, `<prefix>` refers to the type and the first operand is the manipulated instance (e.g., `list.append mylist 42` appends the integer 42 to the specified list reference, `mylist`). Generally, we deliberately limit syntactic flexibility to better support compiler transformations. Recall that HILTI is itself a compiler *target*, and not a language that users are expected to write code for directly.

**Rich Data Types.** While being parsimonious with syntax, we provide HILTI with a rich set of high-level data types relevant to the networking domain. First, HILTI features standard atomic types such as integers, character sequences (with separate types for Unicode strings and raw bytes), floating-point, bitsets, enums, and tuples. In addition, it offers domain-specific types such as *IP addresses* (transparently supporting both IPv4 and IPv6), CIDR-style *subnet masks*, transport-layer *ports*, and *timestamp* / *time interval* types with nanosecond resolution. All these types provide crucial context for type checking, optimization, and data flow/dependency analyses. Second, HILTI offers a set of high-level container types (lists, vectors, sets, maps) that all come with built-in state management support, such as automatic expiration of elements. Iterator types, along with overloaded operators, provide safe generic access to container elements. Further domain-specific types are *overlays* for dissecting packet headers into their components; *channels* for transferring objects between threads; *classifiers* for performing ACL-style packet classification; regular expressions with support for incremental matching and simultaneous matching of multiple expressions; *input sources* for accessing external input (e.g., network interfaces and trace files); *files*; and *timer managers* for maintaining multiple independent notions of time [46].

**Memory Model.** HILTI's memory model is statically type-safe, with containers, iterators, and references pa-

[2] By design, HILTI does not provide any truly global variables visible across threads; exchanging state requires explicit message-passing.

rameterized by type and no explicit cast operator available. A `new` instruction makes dynamic memory allocations explicit. The HILTI runtime automatically garbage-collects objects that have become no longer accessible.

**Flexible Control Flow.** Network traffic analysis is inherently both incremental and parallel: input arrives chunk-wise on concurrent flows. HILTI provides *timers* to schedule function calls for a future time, *snapshots* to save the current execution state for later resumption, *exceptions* for robust error handling, and cooperative multitasking for flexible scheduling (see §3.3). Internally, the main building block for all of these is first-order support for *one-shot continuations* [14]. In addition, *hooks* allow host applications to non-intrusively provide blocks of external code that will run synchronously at well-defined times during execution, with access to HILTI state.

## 3.3 Concurrency Model

A cornerstone of HILTI's design is a domain-specific concurrency model that enables scalable, data-parallel traffic analysis. We base our approach on the observation that networking applications tend to structure their analysis around natural *analysis units* (e.g., per-flow or per-source address), which HILTI leverages to transparently map processing to parallel hardware platforms.

HILTI's concurrency model is a generalization of our earlier, preliminary work on parallelizing the Bro NIDS [46]. There, we found that while generally Bro's custom Turing-complete scripting language does not impose any particular analysis structure, *in practice* tasks tend to be expressed in terms of domain-specific units, such as *per flow* or *per host*. Our analysis found that there is typically no need for state synchronization between the work performed for individual unit instances. For example, code that examines the content of a particular flow rarely needs information about *other* flows. Likewise, a scan detector that tracks connection attempts per source address has no need for correlating counters between sources. Ideally, we would parallelize such analyses by assigning one thread to each individual unit instance (e.g., for the scan detector, one thread for each possible source address, each maintaining a single thread-local counter tracking connection attempts for "its" source).

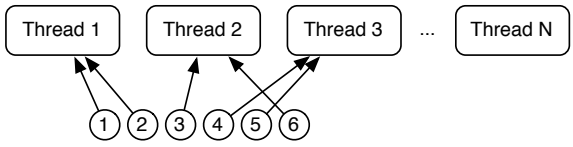
HILTI's concurrency model follows the spirit of this approach. Taking inspiration from Erlang, HILTI uses a large number (10s to 100s) of lightweight, *virtual threads*, each in charge of a set of unit instances. A scheduler steers all processing related to a specific instance to the corresponding thread, which keeps all relevant state in thread-local memory. During execution, the runtime maps virtual threads to *native threads* of the underlying hardware platform via cooperative multitasking.

More concretely, a host application specifies a set of analysis units by defining an *execution context*: a custom composite type consisting of attributes suitable to describe all unit instances. For example, a typical context for a primarily flow-based application is the 4-tuple of addresses and ports. In addition, the application defines a set of scheduling *scopes*, each describing one particular analysis unit as a subset of the context's attributes. With the example 4-tuple-context, scopes might be *flow* (all four attributes), *host pair* (the two addresses), *source* (just the originating address). Finally, the application assigns these scopes to all functions that depend on any of the units, defining their scheduling granularity. At runtime, each virtual thread tracks its current analysis unit as an instance of the execution context, and HILTI's scheduler directs function invocations to the corresponding threads, ensuring that the same target thread will process all invocations triggered by the same unit instance. Internally, a hash of the relevant fields of the current context determines the virtual thread in charge.

```
# Definition of execution context for 4-tuple.
context { addr sa, port sp, addr da, port dp }

# Definition of scopes.
scope Flow   = { sa, sp, da, dp }
scope Source = { sa }

# A function scheduled on a per-flow basis.
void check_http_request(ref<bytes> uri) &scope=Flow
{
    local bool suspicious
    suspicious = regexp.match /param=[aA]ttack/ uri
    ... # Alarm if regular expression matches.
}

# Thread-local map of counters per source address
# (initialization not shown).
global ref<map<addr, int<64>>> conn_attempts

# A function scheduled on a per-source basis.
void count_conn_attempt(addr host) &scope=Source
{
    local int<64> cnt
    # Increase the host's counter (default 0).
    cnt = map.get_default conn_attempts host 0
    cnt = incr cnt
    map.insert conn_attempts host cnt

    ... # Alarm if threshold reached
}
```

Figure 4: HILTI example code for parallelizing a NIDS-style host application.

Figure 4 shows an example demonstrating this process for a NIDS-style application. We define a context, two scopes, and two simple functions: `check_http_request` of scope `Flow`, and `count_conn_attempt` of scope `Source`. We assume that an HTTP protocol parser calls the former for each HTTP request found in the input traffic; the functions then scans the request's URL for suspicious activity. The latter implements a basic scan detector and counts connection attempts per source. Fig-




```
  ... # Get a packet.
  thread.set_context (10.1.1.1,1234/tcp,10.2.2.2,80/tcp)
① thread.schedule check_http_request ("/index.html")
② thread.schedule check_http_request ("/robots.txt")
③ thread.schedule count_conn_attempt (10.1.1.1)
  ... # Get another packet.
  thread.set_context (10.1.1.1,1234/tcp,10.9.9.9,80/tcp)
④ thread.schedule check_http_request ("/index.html")
⑤ thread.schedule check_http_request ("/robots.txt")
⑥ thread.schedule count_conn_attempt (10.1.1.1)
```

Figure 5: Scheduling example for the code in Fig. 4.

ure 5 illustrates the execution. First, the main thread initializes the current execution context with the 4-tuple learned from the header of the currently processed packet, using HILTI's `thread.set_context` instruction. The scheduler now directs the first two invocations of `check_http_request` (①,②) to a single virtual thread as both originate from the same flow. The invocation of `count_conn_attempt` (③) maps to a different thread because an independent unit triggers it (i.e., the source address `10.0.1.1`). When the main thread receives the next input packet, it adapts its context accordingly. The flow-based invocations (④,⑤) now go to a different thread than before, while the target for the source-based invocation (⑥) remains unchanged since the context's originator is still the same. This model guarantees that all map accesses to a particular map index in `count_conn_attempt` will come from a single thread, which can thus use thread-local state to track all connection attempts correctly. Furthermore, the scheduling also *serializes* execution per unit instance, and thus provides a well-defined processing order. The HILTI environment statically enforces correctness of this model by ensuring that functions can schedule only compatible invocations. In the example, `count_conn_attempt` could not call `check_http_request` since its own scope would not be a subset of the callee's.

The scalability of this concurrency model depends crucially on finding sufficient diversity in the analyzed traffic to distribute load and state evenly across threads. Conceptually, HILTI's scheduler works similar to the external, flow-based load-balancing approach that our earlier NIDS cluster work employs [48]. The evaluation of that work, and subsequent experience with several operational installations, show that typical traffic patterns balance well at the flow-level. Moreover, earlier simulations revealed that extending the model to further units provides excellent thread-scalability [46].

While our current prototype focuses on systems with general-purpose commodity CPUs, HILTI's concurrency model also maps well to other parallel architectures. One example is Tilera's many-core CPUs [10], which we examine in §6.2. Another is the massive parallelism offered by today's GPUs. While GPUs are more constrained in their capabilities, we believe that the HILTI compiler has enough analysis context at its disposal that it could automatically identify parts of a HILTI program suitable for outsourcing to a GPU. Finally, HILTI also supports unconstrained manual scheduling across arbitrary virtual threads, allowing for concurrent analysis schemes that do not fit with the context-driven approach.

## 3.4 Profiling

A key challenge for high-volume traffic analysis is assessing and optimizing runtime performance [20]. HILTI supports measuring CPU and memory properties via *profilers* that track attributes such as CPU cycles, memory usage, and cache performance for arbitrary blocks of code. During execution, the HILTI runtime records measured attributes to disk at regular intervals, enabling for example tracking of CPU time spent per time interval [20, 21]. Our HILTI prototype compiler can also insert instrumentation to measure per-function performance, and comes with a tool, `hilti-prof`, that converts the raw profiling records into a human-readable format. HILTI also provides extensive debugging support, which we omit discussing for space reasons.

## 3.5 API

HILTI comes with an extensive C API that offers direct access to all of its data types. In addition, control flow can transfer bi-directionally between external applications and HILTI. C programs call HILTI functions via automatically generated interface stubs, whereas HILTI code can invoke C functions directly. The C interface also integrates exception handling and thread scheduling between HILTI and host application.

A second part of HILTI's API is an extensive Python-based AST interface for constructing HILTI programs in memory. `hiltic` and `hilti-build` are indeed just small wrappers around this API, and host applications can likewise use it for compiling their higher-level analysis specifications into HILTI code. We plan to provide similar interfaces for further languages in the future.

## 3.6 Performance via Abstraction

HILTI's machine model preserves high-level semantics that offer extensive potential for automatic, transparent optimization. We now discuss optimizing individual functionality as well as the generated code in general.

**Optimizing Functionality.** By providing platform-independent abstractions, HILTI enables *transparent* optimization of the underlying implementation. As an example, consider HILTI's `map` data type. While conceptually simple, optimizing its internal hash table for real-time usage requires care to avoid CPU spikes [20]. With HILTI, we can perform such fine-tuning "under the hood" while immediately improving all host applications built on top. Likewise, we are currently working on a JIT compiler for HILTI's regular expressions (similar in spirit to `re2c` [8], yet supporting advanced features such as set matching and incremental operation). Switching to the new JIT version will be an internal change that does not require any modifications to host applications.

HILTI's abstractions also facilitate transparent integration of non-standard hardware capabilities where available. Traditionally, integrating custom hardware elements (such as FPGA-based pattern matchers or network processors) into a networking application has required significant effort for manually adapting the codebase, typically with little potential for reuse in other applications. With HILTI, however, we can adapt the runtime library to leverage the capability transparently while keeping the interface exposed to host applications unchanged. In §6.3 we examine a particular example.

**Optimizing Code.** HILTI's execution model also enables global, whole-program code optimization by providing crucial high-level context for control and dataflow analyses [30]. While we leave a more detailed examination to future work, we believe that state management in particular is an area that can benefit from extensive compile-time optimization. For example, with HILTI's built-in state-management support, we expect the compiler to have sufficient context available to group memory operations for improved cache locality. In addition to purely static optimization schemes, we also plan to integrate approaches driven by runtime profiling [21].

## 4 Applications

To support our claim that HILTI can accommodate a range of common network processing tasks, we built four example host applications: *(i)* a BPF-style packet filter engine; *(ii)* a stateful firewall; *(iii)* a parser generator for network protocols; and *(iv)* a compiler for Bro scripts. All four implementations are early prototypes at this time. While we consider the former two primarily as proofs-of-concept, we intend to further develop the latter two into production systems as HILTI matures.

**Berkeley Packet Filter.** As an initial host application, we implemented a compiler for `tcpdump`-style BPF filters [36]. BPF normally translates filters into code for its custom internal stack machine, which then interprets the code at runtime. Compiling filters into native code via HILTI avoids the overhead of interpreting, enables further compile-time code optimization, and allows extending the filtering capabilities easily.

```
bool filter(ref<bytes> packet) # Ref. to raw data.
  local IP::Header iphdr       # Overlay instance.
  local iterator<bytes> start
  local addr a1, a2
  local bool b1, b2, b3

  bgn = begin packet        # Iterator to 1st byte.
  overlay.attach iphdr bgn  # Attach overlay.

  # Extract fields and evaluate expression.
  a1 = overlay.get iphdr "src"
  b1 = equal a1 192.168.1.1
  a2 = overlay.get iphdr "dst"
  b2 = equal a2 192.168.1.1
  b1 = bool.or b1 b2
  b2 = equal 10.0.5.0/24 a1
  b3 = bool.or b1 b2
  return.result b3
  }
```

Figure 6: Generated HILTI code for the BPF filter `host 192.168.1.1 or src net 10.0.5.0/24`.

Figure 6 shows example HILTI code that our compiler produces for a simple BPF filter. The generated code leverages a predefined *overlay type* for the IP packet header. Overlays are user-definable composite types that specify the layout of a binary structure in wire format and provide transparent type-safe access to its fields while accounting for specifics such as alignment and byte-order.

While our proof-of-concept BPF compiler supports only IPv4 header conditions, adding further `tcpdump` options would be straight-forward. The compiler could also easily go beyond standard BPF capabilities and, e.g., add sampling and stateful filtering [25].

**Stateful Firewall.** Our second proof-of-concept host application is a basic stateful firewall, implemented as a Python script that compiles a list of rules into corresponding HILTI code. To simplify the example, our tool supports only rules of the form `(src-net, dst-net)` $\rightarrow$ `{allow,deny}`, applied in order of specification. The first match determines the result, with a default action of `deny`. In addition, it provides a simple form of stateful matching: when a host pair matches an `allow` rule, the code creates a temporary dynamic rule that will permit all packets in the *opposite direction* until a specified period of inactivity has passed.

Figure 7 shows the code generated for a simple rule set, along with static code that performs the matching. The code leverages two HILTI capabilities: *(i)* the *classifier* data type for matching the provided rules; and *(ii)* a *set* indexed by host pair to record dynamic rules, with a timeout set to expire old entries.

```
### Compiled rule set (net1 -> net2) -> {Allow, Deny}.
### Generated by the application's analysis compiler.

void init_rules(ref<classifier<Rule, bool>> r) {
  # True -> Allow; False -> Deny.
  classifier.add r (10.1.1.1/32, 10.1.0.0/16) False
  classifier.add r (192.168.1.0/24, 10.1.0.0/16) True
  classifier.add r (10.1.1.0/24, *) True
}

### The host applications also provides the following
### static code.

type Rule = struct { net src, net dst } # Rule type.

# Dynamic rules: a set of address pairs allowed to
# communicate. We configure it with an inactivity
# timeout to expire entries automatically (not shown).
global ref<set<tuple<addr, addr>>> state

# Thread-local timer manager controlling expiration
# of dynamic rules (initialization not shown).
global ref<timer_mgr> tmgr

# Function called for each packet, passing in
# timestamp and addresses. Returns true if ok.
bool match_packet(time t, addr src, addr dst) {
  local bool action

  # Advance time to expire old state entries.
  timer_mgr.advance tmgr t

  # See if we have a state entry for this pair.
  action = set.exists state (src, dst)
  if.else action @return_action @lookup

@lookup: # Unknown pair, look up rule.
  try { action = classifier.get rules (src, dst) }
  catch Hilti::IndexError { # No match, default deny.
      return.result False
  }

  if.else action @add_state @return_action

@add_state:     # Add rule allowing opposite direction.
  set.insert state (dst, src)

@return_action: # Return decision.
  return.result action
}
```

Figure 7: HILTI code for firewall example.

While we have greatly simplified this proof-of-concept firewall for demonstration purposes, adding further functionality would be straight-forward. In practice, the rule compiler could support the syntax of an existing firewall system, like `iptables`.

**A Yacc for Network Protocols.** To provide a more complete example, we reimplemented the BinPAC parser generator [40] as a HILTI-based compiler. BinPAC is a "yacc for network protocols": provided with a protocol's grammar, it generates the source code of a corresponding protocol parser. While the original BinPAC system outputs C++, our new version targets HILTI. As we also took the opportunity to clean up and extend the syntax, we nicknamed the new system BinPAC++.

Figure 8 (a) shows a small excerpt of a BinPAC++ grammar for parsing an HTTP request line (e.g., `GET /index.html HTTP/1.1`). In Figure 8 (b) we show the C prototype for the generated parsing function as exposed to the host application, including a struct type corresponding to the parsed PDU. At runtime, the generated HILTI code allocates instances of this type and initializes the individual fields as parsing progresses; see Figure 8 (c) for debugging output showing the process for an individual request. In addition, when the parser finishes parsing a PDU field, it executes callbacks that the host application can optionally provide.

```
const Token      = /[^ \t\r\n]+/;
const NewLine    = /\r?\n/;
const WhiteSpace = /[ \t]+/;

type RequestLine = unit {
    method:  Token;
    :        WhiteSpace;
    uri:     URI;
    :        WhiteSpace;
    version: Version;
    :        NewLine;
};

type Version = unit {
    :       /HTTP\//; # Fixed string as regexp.
    number: /[0-9]+\.[0-9]+/;
};
```

(a) BinPAC++ grammar excerpt for HTTP.

```
struct http_requestline_object {
    hlt_bytes* method;
    struct http_version_object* version;
    hlt_bytes* uri;
    [... some internal fields skipped ...]
};

extern http_requestline_object*
  http_requestline_parse(hlt_bytes *,
                         hlt_exception **);
```

(b) C prototypes generated by HILTI compiler. The host application calls `http_requestline_parse` function to parse a request line.

```
[binpac] RequestLine
[binpac]   method = 'GET'
[binpac]   uri = '/index.html'
[binpac]   Version
[binpac]     number = '1.1'
```

(c) Debugging output showing fields as input is parsed.

Figure 8: BinPAC++ example (slightly simplified).

BinPAC++ provides the same functionality as the original implementation, and we converted parsers for HTTP and DNS over to the new system. Internally, however, we structured BinPAC++ quite differently by taking advantage of HILTI's abstractions. While BinPAC needs to provide its own low-level runtime library for implementing domain-specific data types and buffer management, we now use HILTI's primitives and idioms, which results in higher-level code and a more maintainable parser generator. Leveraging HILTI's flexible control-flow, we now generate fully incremental LL(1)-parsers that postpone parsing whenever they run out of input and trans-

parently resume once more becomes available. In contrast, BinPAC's C++ parsers need to rely on an additional PDU-level buffering layer that often requires additional hints from the grammar writer to work correctly. Finally, while the original version requires that the user write additional C++ code for anything beyond describing basic syntax layout, HILTI enables BinPAC++ to support defining protocol *semantics* as part of the grammar language (e.g., to specify higher-level constraints between separate PDUs). Accordingly, BinPAC++ extends the grammar language with semantic constructs for annotating, controlling, and interfacing to the parsing process, including support for keeping arbitrary state.

**Bro Script Compiler.** Our final application is a compiler for Bro scripts [41]. Different from other, purely signature-based NIDS's, Bro has at its heart a domain-specific, Turing-complete scripting language for expressing custom security policies, as well as prewritten higher-level analysis functionality that ships with the distribution. The language is event-based, with event handlers executing as Bro's C++ core extracts key activity from the packet stream. For example, the internal TCP engine generates a `connection_established` event for any succesful 3-way handshake, passing along meta-data about the corresponding connection as the event's argument. Likewise, Bro's HTTP parser generates `http_request` and `http_reply` events as it parses HTTP client and server traffic, respectively. Many event handlers track state across invocations via global variables, for example to correlate HTTP replies with the earlier requests. Currently, Bro *interprets* all scripts at runtime, which proves to be the primary performance bottleneck in many operational installations.

To demonstrate that HILTI can indeed support such a complex, highly stateful language, we developed a Python-based Bro compiler prototype that translates event handlers into HILTI functions, mapping Bro data types and constructs to HILTI equivalents as appropriate. Using the HILTI toolchain, we can compile these functions into native code that Bro can call directly instead of using its interpreter. While we have not yet implemented the interface between the two systems, we show in §6.1 that the generated code indeed produces the same output as Bro's interpreter when driven by the same event series.

Our compiler supports most features of the Bro scripting language. With HILTI's rich set of high-level data types, we found mapping Bro types to HILTI equivalents straightforward. While Bro's syntax is complex, the compiler can also quite directly lower its constructs to HILTI's simpler register-based language. For example, it transforms loops that iterate over a set of elements into a series of code blocks that use HILTI's operators for type-safe iteration and conditional branching.

There is a small set of Bro language features that the compiler does not yet support, including function pointers (which HILTI does not yet provide) and optional attributes for defining variable properties such as default values for containers and timed expiration of their entries (which HILTI provides but we have not yet implemented for the Bro compiler). Furthermore, the Bro scripting language uses an opaque `any` type to provide functionality like variadic arguments. While by design HILTI's type model does not expose an equivalent to the user, the compiler can either substitute functionality from HILTI's runtime library where available (e.g., HILTI's `printf` equivalent), or use individual versions of a function for all necessary types. In the future, we may add generic functions to HILTI to better support such use cases.

# 5 Implementation

We now discuss the prototype HILTI implementation that we built for studying our design, which consists of a Python-based compiler along with a C runtime library. Unless indicated otherwise, this prototype fully implements all functionality discussed in this paper, and we will release it to the community as open-source software under a BSD-style license. The compiler comes with an extensive test suite of more than 400 units tests. As much of the implementation is a straight-forward application of standard compiler technology, in the following we highlight just some of the more interesting areas.

**Code Generation.** As shown in Figure 1, the compiler, `hiltic`, receives HILTI machine code for compilation. However, instead of generating native code directly, we compile HILTI code into the instruction set of the *Low-Level Virtual Machine (LLVM)* [33], which we leverage for all target-specific code generation. LLVM is an industrial-strength, open-source compiler toolchain that models a low-level yet portable register machine. We also compile HILTI's runtime library into LLVM's representation, using its accompanying C compiler, *clang* [3]. We then link all of the parts together with LLVM's linker, and finally compile the result into a native executable. Alternatively, we could also use LLVM's JIT compiler to perform the last step at runtime.[3]

LLVM provides us with standard, domain-independent code optimizations that are crucial for efficient execution on modern CPUs. Particularly valuable, LLVM performs recursive dead-code elimination across independent link-time units. We leverage this for generating code that can accommodate a range of host applications without sacrificing performance. Consider a HILTI analysis that derives different sets of information

[3] We envision that applications will eventually run the HILTI/LLVM toolchain on-the-fly as they parse a high-level analysis description.

depending on configuration specifics. Some of the information might be expensive to compute and thus the corresponding code should better be skipped if unused. LLVM's global dead code analysis allows the HILTI compiler to initially produce "worst-case code" by including all functionality that a host application *could* use. If it does not, the linker will eventually remove the code and thereby enable further optimizations.

**Execution Model.** `hiltic` implements HILTI's flexible control flow model with first-order continuations by converting function calls into *continuation passing style* (CPS) [13] and generating LLVM code that maintains custom stack frames on the heap. The LLVM compiler performs an optimization pass that replaces CPS-style invocations with simple branch instructions, thereby removing the additional call overhead. We are using a custom calling convention to pass an additional hidden object into all generated functions that provides access to the current thread's state. The generated code relies on automatic garbage collection for all memory management, for which we currently use the well-known Boehm-Demers-Weiser garbage collector [1].

**Runtime Library.** The runtime library implements more complex HILTI data types—such as maps, regular expressions, and timers—as C functions called by the generated LLVM code. LLVM inlines function calls across link-time units and can thus eliminate the extra overhead involved. The runtime library also implements the threading system, mapping virtual threads to native threads and scheduling jobs on a first-come, first-served basis. We use the PAPI library [7] to access CPU performance counters for profiling (see §3.4). Generally, all runtime functionality is fully thread-safe, 64-bit code.

**Limitations.** Our current HILTI implementation is an early prototype that—while already supporting all the discussed functionality—does not yet focus on generating optimized LLVM code. Implementing a compiler that emits high-performance code suitable for processing large volumes of input concurrently and in real-time is a significant engineering task by itself, and as such will entail a significant follow-on effort. Not surprisingly, our initial measurements in §6 show that `hiltic`'s code can be 5–20 times slower than corresponding C code. In §6, we identify two major contributors to this overhead: the prototype's runtime execution model and the garbage collector. The former incurs a significant penalty due to its use of custom stack frames managed on the heap: not only does every function call involve allocating and initializing new memory, but the approach also prevents holding arguments and locals in registers and thus further code optimizations. Regarding memory management, while the Boehm-Demers-Weiser garbage collector is straight-forward to use, it is a conservative collector with well-known limitations [29]. In particular, even when compiled with threading support, it regularly "stops the world" for the collection phase. We did not attempt to further tune the collector.

We are confident that we can solve these limitations with further engineering effort. There are well-known techniques to implement HILTI's functionality efficiently without maintaining a custom stack frames (see, e.g., [17] for an upcoming LLVM-based mechanism); and `hiltic` has the context available to either incorporate a precise concurrent garbage collector, or alternatively use automatic reference counting instead.[4]

# 6 Evaluation

In this section, we assess our design and its prototype implementation in terms of functionality, scalability, and potential for transparent optimization. Given the limitations of the initial implementation (see §5), we cannot present a comprehensive evaluation of the whole system, as much of that would hinge on further engineering efforts to address the current bottlenecks. However, we examine a number of individual aspects, sometimes using simulations as proxies, that demonstrate HILTI's potential as a platform for flexible network traffic analysis.

## 6.1 Functionality

We first verified that the prototype host applications discussed in §4 produce correct results. To keep that validation manageable, we used only a relatively small packet trace captured at the border router of our research institute. With a size of 12GB for 14M packets, it covers about three hours of a normal workday. We first verified that running the compiled filter shown in Figure 7 (using real addresses triggering for 32% of the packets) reported the same number of matches as using `libpcap`. However, the HILTI version needed about 2.7 times more CPU time on a dual quad-core Xeon Linux system. HILTI's profiling showed that the process spent about 65% of its cycles just inside stack frame and glue logic. As `oprofile` [5] furthermore reported that 31% of the total process time is due to the garbage collector, we conclude that the slower execution time is primarily an artifact of our prototype's implementation.

We also confirmed the correctness of the firewall example by comparing with a simple Python script implementing the same functionality. (We skip the details for space reasons.) We verified BinPAC++ by comparing the output of a generated HTTP parser with that of the original C++-based implementation. To that end, we

[4] As Bro's implementation shows, the latter works well for the domain, and Apple's recent reference counting implementation in their LLVM-based Objective C compiler demonstrates using it in LLVM-based applications [2].

instrumented both versions so that they record for every HTTP request its method, path, and version; and for every HTTP reply, the server's status code. Note that for doing so, the parsers need to examine the HTTP exchange in depth to correctly process persistent connections. As both parsers process chunks of application-layer data (not network packets directly), we modified Bro to preprocess the packet trace with its TCP engine and write chunks of in-order application-layer payload to disk. We then wrote wrappers around both HTTP parsers to work from this trace so that they receive their input in the same way as if integrated into the NIDS. Using the same trace as above, we confirmed that the BinPAC++ parser correctly recorded virtually all of the approximately 250,000 requests/replies that the original version extracted. It missed 16, which upon closer inspection however turned out to be incorrectly reported by the old version. Looking at processing times, we saw a major difference: the BinPAC++ parser required about 17 times as much CPU time because the generated recursive-descendant parser takes a large hit from our prototype compiler's high function-call overhead. Accordingly, the HILTI profiler reported that the code spent 49% of its time just doing custom stack management, and *oprofile* credited 51% of the process time to garbage collection.[5] We also ensured correctness of our prototype's scheduler implementation by parallelizing the HTTP analysis on the flow-level, verifying that *(i)* processing indeed distributes across threads, and *(ii)* we see equivalent output with varying numbers of threads.

Finally, we verified that the functionality of the code produced by the Bro script compiler matches that of the original Bro scripts. Here, we focused on HTTP as an example of a complex, stateful protocol that requires Bro to correlate client and server activity at the scripting layer. Bro implements this functionality in two scripts, `http-request.bro` and `http-reply.bro`, that ship as part of its standard library and produce a log file of all HTTP request/reply pairs seen in the input traffic. Doing so entails keeping per-session state that tracks all currently outstanding HTTP requests until the corresponding reply arrives. These two scripts comprise ≈3K lines of Bro script code (including further scripts they import recursively) and use most of the language's capabilities. Recall that the Bro compiler generates functions that correspond to event handlers. For our evaluation, we compiled the two HTTP scripts and then executed the corresponding HILTI functions in isolation. To this end, we recorded the events that Bro's core generates with a given packet trace and converted them into a static series of corresponding HILTI calls to the compiled functions, which we then linked with the compiler's output.

[5] It's a coincidence that these numbers add up to 100%; they are not disjunct as the stack management involves memory operations.

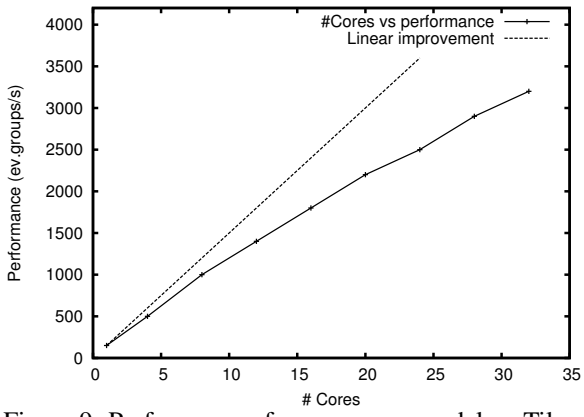


Figure 9: Performance of concurrency model on Tilera.

We disabled some script code that feeds back into Bro's C++ core, such as controlling the BPF filter the system deploys. For tracking time, we continuously updated a global HILTI variable to reflect Bro's "network time", which is the timestamp of the most recent packet. For the comparison, we used one minute of the packet trace described above, generating about 12,500 relevant events.[6] The resulting HTTP log file matched what an unmodified Bro produced from the same input, except for some logging occuring when flushing state at termination that we did not fully emulate.

## 6.2 Scalability

Evaluating the scalability of HILTI's concurrency model is difficult with the current prototype implementation. In initial experiments, we found the garbage-collector to slow down threaded processing to the degree that it was hard to quantify speed improvements realistically. We therefore resorted to simulation for understanding the model's conceptual properties.As a test case, we simulated the processing of Bro event handlers on a 64-core Tilera *TilePro* board [10], assuming the HILTI compilation model discussed in §4. Tilera's processors use a hybrid design, close to both the familiar multi-core paradigm and to proposed many-core architectures [47, 15]. With its large number of parallel computing elements, and high-bandwidth point-to-point links in between, this platform also allows us to examine scalability beyond limits of a conventional commodity architecture.

Currently, however, HILTI code cannot run directly on Tilera, as LLVM does not provide a corresponding backend code generator. We thus used a simulation setup inspired by that in [46]: running Bro 1.5 on an x86 Xeon

[6] We could not use a longer trace as the many function calls produced by the CPS conversion caused our Python-based compiler to require large amounts of main memory. We suspect that there is a memory problem inside the Python-LLVM bindings that we use.

system, we recorded an execution trace of all Bro events generated with a large real-world packet trace. We then wrote a custom simulator in C that, running on the Tilera system, reads the trace and builds *event groups* consisting of all events triggered, directly or indirectly, by the same network packet (note that not all packets trigger events). The simulator then processes each group by distributing the corresponding events across the available cores, similar to how a compiled HILTI program would schedule the work. To determine an event's target core, we annotated all trace records with execution contexts and scopes that would be appropriate if the same task was indeed running on the HILTI scheduler. When receiving an event, a core busy-waited for a time interval proportional to the number of CPU cycles spent during the original x86 run. Finally, to take the bandwidth of Tilera's on-chip network into account, we simulated event parameter passing by exchanging messages of realistic size between communicating cores.

We obtained a realistic workload by recording the original packet trace at a 10 Gbps SPAN port of UC Berkeley's campus border router during a weekday afternoon. To keep the simulation feasible even at slow rates, we limited the execution trace to 1M event groups, corresponding to about 15 seconds of network traffic. We ran our simulation on a *TileExpressPro-20G* device with 64 custom Tilera cores and 4GB of main memory, and configured the device according to manufacturer recommendations for real-time applications, which left up to 32 processors available for simulating the execution. We then performed 9 experiments with 1,4,8,...,32 processors, respectively. In each experiment, we controlled the load by scaling event timestamps proportionally, going from lower to higher event group rates until the device could no longer keep up. Figure 9 shows the relationship between number of processors and maximum event group rate; for reference we also plot the line corresponding to an optimal linear improvement. Although the improvement is less than the optimal, overall performance keeps increasing as we add more processing cores. While we emphasize that our simulation is unlikely to give us realistic estimates on *absolute* throughput—that would require considering the specifics of implementing Bro on the Tilera platform —it does consider scheduling dependencies in the same way an actual system would have to. The results thus show that HILTI's concurrency model is conceptually well-suited to exploit the parallelism inherent in real-world workloads.

## 6.3 Acceleration

One of HILTI's design goals is supporting transparent offloading to specialized hardware, as outlined in §3.6. To evaluate that approach, we examined integrating the recently proposed PLUG architecture [18] into the HILTI runtime. PLUG is a hardware-based parallel lookup engine that accelerates low-level operations frequently performed by packet processing tasks. PLUG takes a deeply pipelined approach that prioritizes throughput over latency. Since highly parallelized applications can best exploit its power, HILTI's concurrency model could provide an ideal fit for the platform.

To validate this hypothesis, we developed a PLUG-based implementation of HILTI's *map* data type, enclosing in a PLUG chip the functionality of an associative container with entry expiration time. Since PLUG is a research project without a commercial implementation, we carried out our evaluation through a custom cycle-accurate functional simulator. The simulator models a simple HILTI application: a flow table that tracks the number of bytes seen per flow. The application deploys per-flow load-balancing across multiple threads, with the PLUG-based data structure servicing lookup operations from multiple threads in parallel.

As a realistic workload for the flow table, we captured 5 minutes of network traffic at the UC Berkeley campus border, and we simulated increasing bandwidths by scaling inter-packet arrival times. The results show that while keeping HILTI's *map* semantics, a PLUG-based flow table can sustain an arrival rate of up to 46M packets/s (corresponding to 15 Gbps assuming 40 byte packets).

In addition to these quantitative results, developing and testing the example application allowed us two interesting observations. First, we were able to map the functionality of HILTI's *map* type to a PLUG implementation without introducing any significant limitations. This implies that one could indeed plug the accelerated container transparently into the HILTI runtime. Second, HILTI's concurrency model—which guarantees serialized execution of all operations accessing the same piece of program state—allowed us to aggressively pipeline part of the implementation without risking data hazards. For example, our `map` implementation interleaves lookup operations even though they are not atomic: they first read an entry and subsequently update its timestamp.

Our simulation assumes sufficient memory to keep the complete map on the PLUG. In practice, the architecture would have only a limited amount of SRAM available (16MB in the current design, allowing for $\approx$ 218,000 concurrent flows; for comparison, our trace would need 128MB if we assume a 1 min inactivity timeout). It is possible however to envision an hybrid scheme where the HILTI runtime transparently keeps high-bandwidth flows on the PLUG, and others in main memory.

## 7 Discussion

**Execution environment.** One might wonder about our choice to build a complete abstract execution environment rather than assemble reusable functionality into, say, a C library. The key here is the additional power that the compilation of HILTI code provides. Typical traffic analysis logic uses a rather small set of conceptual idioms, and it is precisely such domain-specific context that allows modern compiler technology to excel. Any library-based approach would break the tight semantic link between analysis logic and library functionality, preventing optimization schemes from exploiting their full potential [30]. We plan to examine this area further in the future. As one example, consider HILTI's concurrency model. While currently HILTI programs still need to explicitly declare control information in the form of execution context and scopes (see §3.3), the HILTI compiler could likely infer much of that information automatically by tracking their state access patterns.

**Middle-layer.** HILTI serves as a middle-layer between an application's analysis and the low-level code generated for native execution. A different approach would be to provide a platform for *end users* that puts a complete, easy-to-use scripting language at their disposal. However, our primary goal is to enable reuse of functionality across *applications*, and we argue that a single system likely cannot effectively address the needs of both users and applications simultaneously. We envision, however, eventually building a user-level language on top of HILTI for developing higher-level reusable components. Doing so will also make it easier for host applications to provide further custom functionality; currently, one may still need to write a bit of static HILTI code manually for providing additional runtime support (as we did for the firewall example in §4).

**Inline analysis.** Our current HILTI design focuses on passive network analysis, leaving it to host applications to provide active components if desired. However, there are no conceptual limitations preventing HILTI from providing direct support for traffic manipulation inside the forwarding path (e.g., for QoS or normalization [28]). For example, we could interface HILTI to a *Shunt* [50].

## 8 Related Work

By their very nature, existing abstract machine implementations focus on specifics of their respective target domains, and we did not find any that would fit well with the requirements of high-performance network traffic analysis. This includes machine models underlying typical programming languages (e.g., JVM [35], Parrot VM [6], Lua VM [9], Erlang's BEAM/HiPE [43]). However, these mismatches concern primarily high-level, domain-specific functionality, and consequently we leverage an existing *low-level* abstract machine framework, LLVM, in our implementation.

In the networking domain, we find a range of efforts that share aspects with our approach, yet none is providing a similarly comprehensive platform for supporting a wide range of applications. Many could however benefit from using HILTI internally. For example, the C library libnids [4] implements basic building blocks commonly used by NIDS's, paying particular attention to a design robust when facing adversaries and evasion [28]. We envision such libraries to eventually use HILTI for their implementation. Doing so would relieve them from low-level details (e.g., libnids is not thread-safe and has no IPv6 support), and also benefit from a tighter semantic link between host applications and library.

NetShield [34] aims to overcome the fundamentally limited expressiveness of regular expressions by building a custom NIDS engine on top of BinPAC to match more general vulnerability signatures. However, implementing low-level parts of the engine accounts for a significant share of the effort. Using HILTI primitives would be less time-consuming and also enable other applications to share the developed functionality.

The Click modular router [31] allows users to compose a software router from elements that encapsulate predefined primitives, such as IP header extractors, queues, and packet schedulers. The connected elements form a directed graph where packets travel along the edges, and a custom configuration language maps such graph specifications to C++ code. Click configurations could alternatively compile into HILTI.

RouteBricks [22] is a multi-Gbps distributed software router that scales both across cores and machines. To optimize intra-node parallelization, RouteBricks uses techniques akin to HILTI's concurrency model: per-core packet queues enable an efficient lock-free programming model with good cache performance. HILTI provides for such per-flow analysis (within a single system) by routing related packets to the same thread. HILTI's concurrency model is more general, though, and allows other scheduling strategies as well. Moreover, its abstract machine model supports a variety of applications.

NetPDL [44] is an XML-based language to describe the structure of packet headers. It decouples protocol parsing code from protocol specifics. The language supports fixed and variable-length protocol fields as well as repeated and optional ones. While NetPDL takes a conceptually different approach than BinPAC, it uses similar building blocks and could thus leverage HILTI.

Xplico [11] is a network forensic tool written in C that ships with protocol decoders and manipulators, including web chat protocols inside HTTP. While powerful in its capabilities, the system's implementation demonstrates

the challenge of implementing the required low-level functionality. The HTTP protocol decoder, for example, reassembles HTTP payload by writing all packet content into per-flow *files* on disk, which are then re-read by higher-level analyzers for further analysis. HILTI makes it easier to implement such functionality efficiently.

Software-defined networking (SDN) separates a network's device control and data planes, allowing operators to program routers and switches. OpenFlow [37] provides a vendor-agnostic interface to such functionality, and a number of higher-level languages [26, 32, 23, 38] use it to control compatible hardware devices. If we added an OpenFlow interface to HILTI's runtime, it could drive the software component of such systems.

NetVM [39] compiles Snort rules into a custom intermediary representation, and from there just-in-time into native code. It routes packets through a graph of connected network elements, each of which features a stack-based processor, private registers, and a memory hierarchy. NetVM's functionality is however much more limited than what HILTI provides: its primary goal is to achieve portability of signature matching, and, as such, its intermediary code is lower-level and more narrowly focused on the task at hand. Moreover, HILTI's compilation into LLVM code enables exploiting that toolchain's full optimization support, whereas it appears difficult to optimize across NetVM elements.

Similar to our example in §4, Linux recently added support for JIT compiling BPF expressions into native (x86) assembly [16]. FreeBSD 7 also includes experimental BPF JIT support (x86 and amd64). HILTI provides a portable, platform-independent approach by employing LLVM for native code generation.

Finally, there is a large body of work on accelerating parts of the network traffic analysis pipeline with custom hardware elements, targeting for example pattern matching (e.g., [45, 24]), parallelization on GPUs (e.g., [49, 12, 27]), robust TCP stream reassembly [19], and high-speed lookup tables such as PLUG; see §6.3). HILTI's design allows to transparently offload specific computations to specialized hardware when available.

## 9 Conclusion

We present the design of HILTI, a novel platform for concurrent, stateful network traffic analysis. HILTI is a middle-layer located between a host application and the hardware platform that executes the analysis. Our motivating observation is the fact that many networking applications share a large set of functionality, yet typically reimplement nearly all of it from scratch each time, often running into problems that our community has already solved numerous times before. HILTI aims to bridge that gap by providing common functionality to applications that they can use to express tasks in higher-level terms.

We have developed a prototype compiler that implements all of HILTI's initially targeted functionality, including rich domain-specific data types, automatic memory management, flexible control flow, scalable parallelization, profiling and debugging support, and an extensive API for host applications. We developed four example applications that demonstrate HILTI's ability to support a range of typical network analyses. Through transparent optimization and integration of non-standard hardware elements, HILTI has the potential to fully exploit the capabilities of its underlying hardware platform.

Our goal is to further develop HILTI into a platform suitable for operational deployment in large-scale network environments. While our prototype is not yet there, we are confident that further engineering effort can overcome the current bottlenecks. In addition, we also envision HILTI supporting future research by greatly simplifying the prototyping of new traffic analysis approaches.

## References


[1] A garbage collector for C and C++. `http://www.hpl.hp.com/personal/Hans_Boehm/gc`.

[2] Automatic Reference Counting. `http://clang.llvm.org/docs/AutomaticReferenceCounting.html`.

[3] clang. `http://clang.llvm.org`.

[4] Libnids. `http://libnids.sourceforge.net`.

[5] OProfile. `http://oprofile.sourceforge.net`.

[6] Parrot Virtual Machine. `http://docs.parrot.org`.

[7] Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/`.

[8] re2c. `http://re2c.org`.

[9] The Programming Language Lua. `http://www.lua.org`.

[10] Tilera TILEPro64 Processor. `http://www.tilera.com/products/TILEPro64.php`.

[11] Xplico. `http://www.xplico.org`.

[12] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. In *Proc. ACM SIGCOMM*, 2010.

[13] A. W. Appel. *Compiling with Continuations*. CUP, 1992.

[14] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing Control in the Presence of One-Shot Continuations. In *Proceedings of the ACM SIGPLAN*, 1996.

[15] D. Burger et al. Scaling to the End of Silicon with EDGE Architectures. *Computer*, 37:44–55, July 2004.

[16] J. Corbet. A JIT for packet filters. `http://lwn.net/Articles/437981/`.

[17] S. Das. Segmented Stacks in LLVM. `http://www.google-melange.com/gsoc/project/google/gsoc2011/sanjoyd/13001`.

[18] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam. PLUG: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-Speed Routers. *SIGCOMM Comput. Commun. Rev.*, 39:207–218, August 2009.

[19] S. Dharmapurikar and V. Paxson. Robust TCP Stream Reassembly in the Presence of Adversaries. In *USENIX Security*, 2005.

[20] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational Experiences with High-Volume Network Intrusion Detection. In *ACM CCS*, Oct. 2004.

[21] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the Resource Consumption of Network Intrusion Detection Systems. In *Proc. RAID*, 2008.

[22] K. Fall, G. Iannaccone, M. Manesh, S. Ratnasamy, K. Argyraki, M. Dobrescu, and N. Egi. RouteBricks: Enabling General Purpose Network Infrastructure. *SIGOPS Operating Systems Review*, 45:112–125, February 2011.

[23] N. Foster et al. Frenetic: A High-Level Language for OpenFlow Networks. In *Proc. PRESTO*, 2010.

[24] R. Franklin, D. Carver, and B. Hutchings. Assisting network intrusion detection with reconfigurable hardware. In *Proceedings from Field Programmable Custom Computing Machines*, 2002.

[25] J. M. Gonzalez and V. Paxson. Enhancing Network Intrusion Detection With Integrated Sampling and Filtering. In *Proc. RAID*, 2006.

[26] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 38:105–110, July 2008.

[27] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proc. ACM SIGCOMM*, 2010.

[28] M. Handley, C. Kreibich, and V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proc. USENIX Security*, 2001.

[29] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook*. Cambridge University Press, 2011.

[30] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., 2002.

[31] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18:263–297, August 2000.

[32] T. Koponen et al. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *Proc. USENIX OSDI*, 2010.

[33] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Intl. Symposium on Code Generation and Optimization*, 2004.

[34] Z. Li et al. NetShield: Massive Semantics-Based Vulnerability Signature Matching for High-Speed Networks. In *Proc. ACM SIGCOMM*, 2010.

[35] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Prentice Hall, 1999.

[36] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. USENIX Winter 1993 Conference*, pages 259–270, January 1993.

[37] N. McKeown et al. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, 2008.

[38] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. July 2011. Draft.

[39] O. Morandi, G. Moscardi, and F. Risso. An Intrusion Detection Sensor for the NetVM Virtual Processor. In *Proc. ICOIN*, 2009.

[40] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *ACM Internet Measurement Conference*, 2006.

[41] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[42] V. Paxson, K. Asanovic, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In *Proceedings of the USENIX Hot Security Workshop*, August 2006.

[43] M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. In *Proc. FLOPS*, 2002.

[44] F. Risso and M. Baldi. NetPDL: An Extensible XML-based Language for Packet Header Description. *Computer Networks*, 50:688–706, April 2006.

[45] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *Proc. IEEE FCCM*, Apr. 2001.

[46] R. Sommer, V. Paxson, and N. Weaver. An Architecture for Exploiting Multi-Core Processors to Parallelize Network Intrusion Prevention. *Concurrency and Computation: Practice and Experience*, 21(10):1255–1279, 2009.

[47] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proc. IEEE/ACM MICRO*, 2003.

[48] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proc. RAID*, 2007.

[49] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proc. RAID*, 2008.

[50] N. Weaver, V. Paxson, and J. M. Gonzalez. The Shunt: An FPGA-Based Accelerator for Network Intrusion Prevention. In *Proc. ACM FPGA*, February 2007.