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

Robin Sommer[1,2], Matthias Vallentin[3], Lorenzo De Carli[4], Vern Paxson[1,3]

[1]International Computer Science Institute   [2]Lawrence Berkeley National Laboratory   [3]UC Berkeley
[4]University of Wisconsin-Madison

ABSTRACT

When developing networking systems such as firewalls, routers, and intrusion detection systems, one faces a striking gap between the ease with which one can often describe a desired analysis in high-level terms, and the tremendous amount of low-level implementation details that one must still grapple with to come to a robust solution. We present HILTI, a *platform* that bridges this divide by providing to application developers much of the low-level functionality, without tying it to a specific analysis structure. HILTI consists of two parts: *(i)* an *abstract machine model* that we tailor specifically to the networking domain, 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. We have developed a prototype of a HILTI compiler toolchain that fully implements the design's functionality, and ported exemplars of networking applications to the HILTI model to demonstrate the aptness of its abstractions. Our evaluation of HILTI's functionality and performance confirms its potential to become a powerful platform for future application development.

1. INTRODUCTION

Deep, stateful network packet inspection represents a crucial building block for applications that analyze network traffic. However, when developing 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 come to an efficient and robust implementation. When applications reconstruct a 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 conservative fail-safe processing. Despite such implementation challenges, our community sees little reuse of existing, well-proven functionality across applications. While individual projects invest significant resources into optimizing their particular implementation, new efforts can rarely leverage the accumulated experience that such systems have garnered through years of deployment. Not only do the they end up building much of same functionality from scratch each time, but they also tend to fall into the same pitfalls that others had to master before.

In this work we set out to overcome this situation. We present a novel *platform* for building network traffic analysis applications that provides 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* that we tailor specifically to the networking domain, 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. Our abstract machine model has at its core a *high-level intermediary language for traffic inspection (HILTI)* that provides rich data structures, powerful control flow primitives, extensive concurrency support, a secure memory model protecting against unintended control and data flows, and potential for domain-specific optimizations that include transparent integration of non-standard hardware capabilities. Conceptually, HILTI provides a *middle-layer* situated between the operating system and a *host application*, operating invisibly to the application's end-users who do not directly interact with it. An application leverages HILTI by *compiling* its analysis/functionality from its own high-level description (like a firewall's rules or a NIDS's signature set) into HILTI code. HILTI's compiler then translates it further down into native code.

We have developed a prototype of the HILTI compiler toolchain that fully supports all of the design's functionality, and we have ported a set of example networking applications to the HILTI machine model that demonstrate the aptness of its abstractions, including the BinPAC protocol parser generator [34] and Bro's scripting language [35]. Even though performance does not yet represent a particular focus of the prototype's code generator, our evaluation with real-world network traffic shows that already the compiled code executes with generally similar performance as native implementations, with the potential to significantly outperform them in the future as we further optimize the toolchain.

As one of HILTI's overarching objectives we aim to build a platform that integrates much of the knowledge that the networking community has collected over decades into a sin-

Traffic Analysis Building Blocks

Domain-specific data types	State management	Concurrent analysis	Real-time performance	Robust execution	High-level components

HILTI Environment

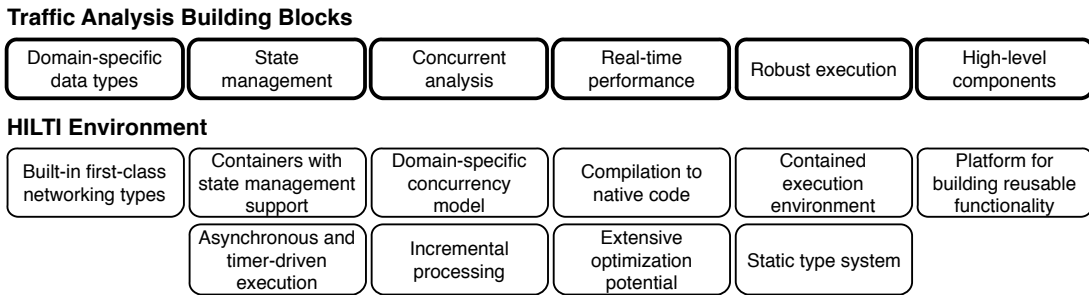| Built-in first-class networking types	Containers with state management support	Domain-specific concurrency model	Compilation to native code	Contained execution environment	Platform for building reusable functionality
	Asynchronous and timer-driven execution	Incremental processing	Extensive optimization potential	Static type system	

Figure 1: Building blocks of network traffic applications and how HILTI supports them.

gle framework for applications to build upon. While in this work we focus on the low-level machine model, we envision HILTI to eventually ship 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 work that others have previously implemented.

We structure the remainder of this paper as follows. We motivate our work in §2 by examining the potential for sharing functionality across networking applications. We present HILTI's design in §3 and showcase four example applications in §4. We discuss our implementation in §5, and then evaluate HILTI in §6. We discuss some broader aspects in §7, and summarize related work in §8. We conclude in §9.

2. SHARING FUNCTIONALITY

Internally, different types of networking applications—packet filters, stateful firewalls, routers, switches, intrusion detection systems, network-level proxies, and even OS-level packet processing—all exhibit a similar structure that builds on a common set of domain-specific idioms and components.[1] Implementing such standard functionality is not rocket science. However, experiences with developing robust and efficient monitoring systems reveal that coming to correct and memory-safe code quickly proves challenging—much more than one might intuitively expect. It is, hence, unfortunate that in contrast to other domains where communities have developed a trove of reusable standard functionality (e.g., HPC or cryptography), we find little sharing across networking systems—not 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 standard data structures with state management, support for asynchronous execution, logic for

---

[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 could benefit from HILTI as well they tend to have different characteristics that exceed the scope here.

discerning IPv4 and IPv6 addresses, and protocol inspection. We also compared the source code of the three major open-source NIDS implementations (Bro, Snort, and Suricata), and we found neither any significant code reuse across these systems, nor leveraging of much third-party functionality. `libpcap` and `libz` are the only external libraries to which they all link. In addition, Snort and Suricata both leverage `PCRE` for regular expression matching, while Bro implements its own engine. Indeed, in a panel discussion at RAID 2011 all three projects acknowledged the lack of code reuse, attributing it to low-level issues concerning program structures and data flows.

From our source code analysis we identify a number of common building blocks for networking applications, as illustrated in Figure 1:

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, network prefixes, 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. However, managing that state in real-time requires effective expiration strategies [14]. HILTI provides container types with built-in state management, and timers to schedule processing asynchronously into the future, e.g., for customized cleanup tasks or time-driven logic.

Concurrent Analysis. High-volume network traffic exhibits an enormous degree of inherent parallelism [36], and applications need to multiplex their analyses across potentially tens of thousands of data flows, either inside a single thread or parallelized across multiple CPUs. High-performance applications employ the latter only sparingly today, as it remains challenging to parallelize stateful analysis efficiently across threads while maintaining linear scaling with the workload size. HILTI supports both forms of parallelism by *(i)* enabling transparent incremental processing, and *(ii)* providing a concurrency model that employs cooperative multitasking to supply applications with a large numbers of lightweight threads with well-defined semantics.

Real-time Performance. With 10 Gbps links now standard even in medium-sized environments, applications deal with enormous packet volumes in real-time. In addition to supporting parallel processing, HILTI compiles analy-
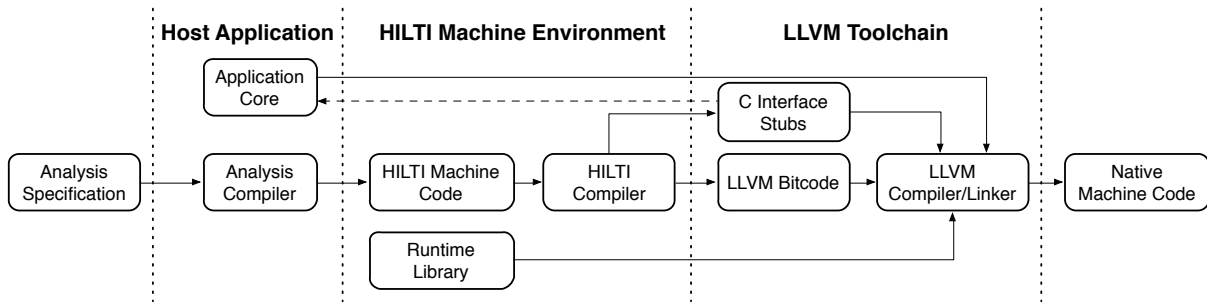
Figure 2: Workflow for using HILTI.

```
# 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.

ses into native executables with the potential for extensive domain-specific code optimization.

Robust/Secure Execution. Networking applications process untrusted input: attackers might attempt to mislead a system, and—more mundanely— real-world traffic contains plenty "crud" [35] not conforming to any RFC. While writing robust C code remains notoriously difficult, HILTI's abstract machine model provides a contained, well-defined, and statically typed environment that, for example, prevents code injection attacks.

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

3. HILTI ABSTRACT MACHINE MODEL

The heart of HILTI consists of an abstract machine model that we tailor to network traffic analysis. In the following we discuss HILTI's design in more detail. For brevity we will use the name HILTI for both the abstract machine model itself as well as for the framework that implements it, i.e., the compiler toolchain and runtime library.

3.1 Workflow

Figure 2 shows the overall workflow when using HILTI. A *host application* leverages HILTI for providing its functionality. Typically, the host application has a user-supplied *analysis specification* that it wants to instantiate; e.g., the set of filtering rules for a firewall, or the set of signatures for a NIDS. The application provides a custom *analysis compiler* that translates its specifications into HILTI machine code, either in the form of text files or, more conveniently, as an

in-memory AST representation it creates via a corresponding C++ API that HILTI provides. In either case the *HILTI compiler* then compiles that input into *bitcode* for *LLVM* (the Low-Level Virtual Machine [28]), which we leverage for all target-specific code generation. The HILTI compiler also generates a set of *C stubs* for the host application to interface with the resulting code. Finally, LLVM combines compiled code, stubs, runtime library, and the host application into a single unit of native machine code for execution, either statically or just-in-time at startup (JIT). Our prototype includes two tools, `hiltic` and `hilti-build`, which employ this workflow to compile HILTI code into native objects and executables, respectively. Figure 3 shows an example compiling a trivial HILTI program into a static binary. Alternatively, `hiltic` can JIT-execute the source directly.

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 further C-level functions called out to as necessary as if using an external library; Figure 3 demonstrates this model by defining the `Main::run()` as the entry point. Alternatively, the host application itself can drive execution and leverage HILTI-based functionality by calling the generated C stubs on demand (e.g., a NIDS might feed payload into a HILTI-based protocol parser; see §4).

3.2 Execution Model

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 exist primitives to define functions, custom data types, and local and thread-local variables (but no truely global; see below). By convention, mnemonics have the form `<prefix>.<operation>`, where the same `prefix` indicates a related set of functionality; Table 1 summarizes the available groups. For data types in particular, `<prefix>` refers to the type and the first operand to the manipulated instance, e.g., `list.append l 42`, appends the integer 42 to the specified list `l`. In total HILTI currently offers about 200 instructions (counting instructions overloaded by their argument types only once). Generally, we deliberately limit syntactic flexibility to better support compiler transforma-

tions because HILTI mainly acts as compiler *target*, and not a language that users 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 statically typed 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 come with built-in state management support that automatically expires elements according to a given policy. Iterators, along with overloaded operators, provide type-safe generic access to container elements. Further domain-specific types include *overlays* for dissecting packet headers into their components; *channels* for transferring objects between threads; *classifiers* for performing ACL-style packet classification; *regular expressions* supporting incremental matching and simultaneous matching of multiple expressions; *input sources* for accessing external input (e.g., network interfaces and trace files); *timer managers* for maintaining multiple independent notions of time [41]; and *files*.

Memory Model. HILTI's memory model is statically type-safe, with containers, iterators, and references parameterized by type. A `new` instruction makes dynamic allocations explicit. The HILTI runtime automatically garbage-collects objects that have become no longer accessible.

Control Flow and Concurrency. For flexible control flow, HILTI provides *timers* to schedule function calls to the future; *closures* for capturing function calls; *exceptions* for robust error handling; and *hooks* for allowing host applications to provide non-intrusive callbacks that run synchronously at well-defined times.

HILTI also supports two styles of concurrency. First, within a single hardware thread, one can multiplex analyses by switching arbitrarily between stacks using co-routines. Consider protocol analysis where the host application incrementally feeds chunks of payload into the parsing code, switching between different sessions as their packets arrive. When the host's C code calls for the first time the HILTI parsing function for a session, HILTI internally instantiates a *fiber* object for the call's execution that can record its state, and then proceeds with processing the data. Once it reaches the end of the currently available chunk, it suspends execution by freezing the stack and copying the CPU's registers into the fiber. Later, when new payload arrives for that session, the application resumes parsing by reactivating the fiber. HILTI reinstates the frozen stack and continues where it left off. Compared to traditional implementations—which typically maintain per-session state machines manually that

Functionality	Mnemonic	Functionality	Mnemonic
Bitsets	`bitset`	Packet I/O	`iosrc`
Booleans	`bool`	Packet classification	`classifier`
CIDR masks	`network`	Packet dissection	`overlay`
Callbacks	`hook`	Ports	`port`
Closures	`callable`	Profiling	`profiler`
Channels	`channel`	Raw data	`bytes`
Debug Support	`debug`	References	`ref`
Doubles	`double`	Regular Expressions	`regexp`
Enumerations	`enum`	Strings	`string`
Exceptions	`exception`	Structs	`struct`
File I/O	`file`	Time Intervals	`interval`
Flow Control	(No joint prefix)	Timer Management	`timer_mgr`
Hashmaps	`map`	Timers	`timer`
Hashsets	`set`	Times	`time`
IP Addresses	`addr`	Tuples	`tuple`
Integers	`int`	Vectors/Arrays	`vector`
Lists	`list`	Virtual Threads	`thread`

Table 1: HILTI's main instruction groups.

record the current state—this model remains transparent to the analysis code and hence simplifies its structure.

Second, HILTI provides threads for distributing an analysis concurrently across CPU cores. We employ an Erlang-style threading model that provides an application with a large supply of lightweight *virtual threads*, which a runtime scheduler then maps to a small number of hardware threads via cooperative multi-tasking. HILTI identifies threads with 64-bit integer IDs: The instruction `thread.schedule foo("abc") 123456` schedules an asynchronous invocation of the function `foo` with one string argument `abc` to thread number `123456`. The ID-based model maps directly to hash-based load-balancing schemes that networking applications commonly deploy for parallel traffic analysis (e.g., Suricata [5] and Bro [42]). For example, to distribute flow processing across threads, one would hash the flow's 5-tuple into an integer and then interpret that value as the ID of the virtual thread to assign the corresponding analysis to. As processing within each virtual thread proceeds sequentially, this approach implicitly serializes all computation relating to a single flow and thus obviates the need for further intra-flow synchronization. Per [41], we find typical analysis tasks amendable to this model.

HILTI's execution model prevents data races and low-level dead-locks by design: virtual threads cannot share state directly. In particular, HILTI does not provide global variables visible across threads. Instead, each virtual thread receives its own set of *thread-local* variables for recording state related to its processing. Exchanging global state requires explicit message passing, either by connecting threads with HILTI's thread-safe channel data type, or by scheduling tasks to a target thread and passing the relevant information as arguments. [2] In either case, the runtime deep-copies all mutable data so that the sender will not see any modifications that the receiver may make. This strict model of data isolation enables reliable concurrent execution because it encourages value semantics as opposed to complicated data dependencies that would require locks to synchronize access.

---

[2]Note that the interpretation of HILTI's `global` keyword in code examples below is "a variable global to the current virtual thread".

3.3 Profiling & Debugging

A key challenge for high-volume traffic analysis is assessing and optimizing runtime performance [14]. 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, e.g., enabling tracking of CPU time spent per time interval [14, 15]. The HILTI compiler can also insert instrumentation to profile at function granularity.

3.4 Host Application API

HILTI comes with an extensive C API that offers host applications direct access to its data types. In addition, control flow can transfer bi-directionally between applications and HILTI. C programs call HILTI functions via the C stubs, and HILTI code can invoke arbitrary C functions. The API integrates exception handling, timer management, fiber resumption, and thread scheduling between HILTI and application.

A second part of HILTI's API is a C++ AST interface for constructing HILTI programs in memory. `hiltic` and `hilti-build` are in fact just wrappers around this API, which host applications can likewise employ to compile their analysis specifications into HILTI code. Combining AST and JIT interfaces enables applications to go all the way from user-level specification to native code on the fly.

4. APPLICATION EXEMPLARS

We now develop four applications as examples to illustrate HILTI's ability to accommodate a wide range of common network processing tasks: *(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. We have implemented all four as prototypes. While we consider the former two primarily proof-of-concepts, the latter two represent realistic and fully functional applications that we intend to further improve as HILTI matures.

Berkeley Packet Filter.

As an initial simple application to explore, we implemented a compiler for BPF [30]. BPF traditionally translates filters into code for its custom internal stack machine, which it then interprets at runtime. Compiling filters into native code via HILTI avoids the overhead of interpreting, enables further compile-time code optimization, and facilitates easy extension of the filtering capabilities in the future.

Figure 4 shows HILTI code that our compiler produces for a simple BPF filter. The generated code leverages a HILTI *overlay type* for parsing IP packet headers. 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 endianness. While our proof-of-concept BPF compiler supports only IPv4 header conditions, adding further `BPF` features would be straight-forward. The com-

```
type IP::Header = overlay {
 # <name>: <type> at <offset> unpack <format> [(bits)]
  version: int<8> at 0 unpack UInt8InBigEndian (4, 7),
  hdr_len: int<8> at 0 unpack UInt8InBigEndian (0, 3),
  [...]
  src:     addr    at 12 unpack IPv4InNetworkOrder,
  dst:     addr    at 16 unpack IPv4InNetworkOrder
}

bool filter(ref<bytes> packet) { # Input: raw data.
  local addr a1, a2
  local bool b1, b2, b3

  # Extract fields and evaluate expression.
  a1 = overlay.get IP::Header src packet
  b1 = equal a1 192.168.1.1
  a1 = overlay.get IP::Header dst packet
  b2 = equal a2 192.168.1.1
  b1 = or b1 b2
  b2 = equal 10.0.5.0/24 a1
  b3 = or b1 b2
  return b3
}
```

Figure 4: Generated HILTI code for the BPF filter
`host 192.168.1.1 or src net 10.0.5.0/24.`

piler could also go beyond standard BPF capabilities and, for example, add stateful filtering [23].

Stateful Firewall.

Our second proof-of-concept host application is a basic stateful firewall, implemented as a Python host application 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, we provide for 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 5 shows the code generated for a simple rule set, along with additional 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. While we have simplified this proof-of-concept firewall for demonstration purposes, adding further functionality would be straight-forward. In practice, the rule compiler could directly 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 [34] as a HILTI-based compiler. BinPAC is a "yacc for network protocols": given 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 syntax and functionality, we nicknamed the new system BinPAC++.

Figure 6 (a) shows an excerpt of a BinPAC++ grammar for

```
### 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 (10.3.2.1/32,  10.1.0.0/16) True
  classifier.add r (10.12.0.0/16, 10.1.0.0/16) False
  classifier.add r (10.1.6.0/24,  *) True
  classifier.add r (10.1.7.0/24,  *) True
}

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

# Data type for a single filter rule.
type Rule = struct { net src, net dst }

# The classifier storing the rules.
global ref<classifier<Rule, bool>> rules

# Dynamic rules: address pairs allowed to communicate.
global ref< set< tuple<addr, addr> > > dyn

# Function to initialize classifier at startup.
void init_classifier() {
  rules = new classifier<Rule, bool> # Instantiate.
  call init_rules(rules)             # Add rules.
  classifier.compile rules           # Freeze/finalize.

  # Create set for dynamic state with timeout of 5 mins
  # of inactivity.
  dyn = new set<tuple<addr, addr>>
  set.timeout dyn ExpireStrategy::Access interval(300)
}

# 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 HILTI's global time. This will expire
  # inactive entries from the state set.
  timer_mgr.advance_global t

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

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

  if.else action add_state return_action

add_state: # Add rules to allow both directions.
  set.insert dyn (src, dst)
  set.insert dyn (dst, src)

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

Figure 5: HILTI code for firewall example.

parsing an HTTP request line (e.g., GET /index.html HTTP/1.1). In Figure 6 (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 protocol data unit (PDU). At runtime, the generated HILTI code allocates instances of this type and sets the individual fields as parsing progresses, as the debugging output in Figure 6 (c) shows for an individual request. When the parser finishes with a field, it executes any callbacks (hooks) that the host application specified for that field. If the grammar does not include hooks, the host application can still extract

```
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;
    hlt_bytes* uri;
    struct http_version_object* version;
    [... 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 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 6: BinPAC++ example (slightly simplified).

the parsed data from the returned struct (as Figure 6 (b) exemplifies) after HILTI finishes processing the type.[3]

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 could structure BinPAC++ quite differently by taking advantage of HILTI's abstractions. While the original BinPAC provides 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 can now generate fully incremental LL(1)-parsers that postpone parsing whenever they run out of input and transparently resume once more becomes available. In contrast, the C++ parsers of the original BinPAC need to manually manage the process and further rely on an additional PDU-level buffering layer that often requires additional hints from the grammar writer to work correctly. Finally, while the original BinPAC requires the user to write additional C++ code for any logic

---

[3] An advantage of using hooks is that HILTI could optimize away the parsing of unused fields and also avoid further unnecessary data copying (not implemented yet).

beyond describing basic syntax layout, BinPAC++ extends the grammar language with semantic constructs for annotating, controlling, and interfacing to the parsing process, including support for keeping arbitrary state, by compiling them to corresponding HILTI code.

BinPAC++ remains independent of a specific host application because HILTI generates external C interfaces for the compiled parsers. We demonstrate this functionality by extending Bro to both drive the parsing and use the results, just as it does with its built-in protocol parsers. As Bro decodes protocols, it executes *event handlers* written in Bro's custom scripting language. For example, the internal TCP parser generates a `connection_established` event for any successful three-way handshake, passing along meta-data about the corresponding connection as the event's argument. Similarly, Bro's HTTP parser generates `http_request` and `http_reply` events for client and server traffic, respectively. To define the events that BinPAC++ will generate for Bro, we add additional "event configuration files". Figure 7 showcases the interaction between the two systems: (a) shows a basic BinPAC++ grammar to parse SSH banners; the event configuration file in (b) tells Bro to trigger an `ssh_banner` event whenever the generated code finishes parsing an `SSH::Banner` unit; and (c) shows corresponding Bro script code that implements a simple handler for that event. Finally, (d) demonstrates the usage: Bro loads the event configuration file from the command line, pulls in the corresponding BinPAC++ grammar, compiles the grammar into HILTI parsing code, and generates internal HILTI glue code that at runtime interfaces back to Bro's event engine for triggering the defined events.

Bro Script Compiler.

Our final application implements a compiler for Bro scripts [35]. Unlike purely signature-based NIDSs, Bro's domain-specific, Turing-complete scripting language decouples the system from any specific analysis model. In addition to detection tasks, Bro uses its language also for higher-level analysis logic in the form of library functionality that ships with the distribution. For example, most protocol-specific scripts perform per-session state-tracking, such as the HTTP analysis correlating replies with earlier requests.

To demonstrate that HILTI can indeed support such a complex, highly stateful language, we developed a Bro plugin that translates all loaded scripts scripts into corresponding HILTI logic, mapping Bro constructs and data types to HILTI equivalents as appropriate. The plugin executes the HILTI toolchain just-in-time, generating native code at startup. When Bro generates events, it triggers the HILTI code instead of going through its standard script interpreter. With HILTI's rich set of high-level data types we generally found mapping Bro types to HILTI equivalents straightforward. While Bro's syntax is complex, the compiler can generally directly convert its constructs to HILTI's simpler register-based language. Figure 8 shows a simple exam-

```
module SSH;

export type Banner = unit {
    magic   : /SSH-/;
    version : /[^-]*/;
    dash    : /-/;
    software: /[^\r\n]*/;
};
```

(a) BinPAC++ grammar for SSH banners in `ssh.pac2`.

```
grammar ssh.pac2; # BinPAC++ grammar to compile.

# Define the new parser.
protocol analyzer SSH over TCP:
    parse with SSH::Banner, # Top-level unit.
    port 22/tcp,            # Port to trigger parser.

# For each SSH::Banner, trigger an ssh_banner() event.
on SSH::Banner
    -> event ssh_banner(self.version, self.software);
```

(b) Event configuration file in `ssh.evt`.

```
event ssh_banner(version: string, software: string) {
    print software, version;
    }
```

(c) Bro event handler in `ssh.bro`.

```
# bro -r ssh.trace ssh.evt ssh.bro
OpenSSH_3.9p1, 1.99
OpenSSH_3.8.1p1, 2.0
```

(d) Output with a single SSH session (both sides).

Figure 7: Bro/BinPAC++ interface example.

ple compiling two event handlers tracking the server addresses of all established TCP connections. As the example shows, the compiler translates Bro's event handler into HILTI hooks (which are, roughly, functions with multiple bodies that all execute upon invocation).

Our prototype compiler supports most features of the Bro scripting language.[4] Specifically, it supports running Bro's default HTTP and DNS analysis scripts, which we use as representative case studies in our evaluation (see §6). The HTTP and DNS scripts generate extensive logs of the corresponding protocol activity, correlating state across request and reply pairs, plus (in the case of HTTP) extracting and identifying message bodies.

5. IMPLEMENTATION

We now discuss our prototype HILTI implementation, which consists of a C++-based compiler along with a C runtime library. In addition, we have implemented the four application scenarios we discuss in §4 in the form of *(i)* Python scripts for the two proof-of-concepts (BPF and Firewall); *(ii)* a C++ compiler to translate BinPAC++ grammars into HILTI; and *(iii)* a Bro plugin in C++ that (a) provides a run-

---

[4]We currently exclude some rarely used constructs as well as a small set of advanced functionality that does not have direct HILTI equivalents yet. In particular, the compiler lacks support for Bro's `when` statement, which triggers script code asynchronously once a specified global condition becomes true. We plan to add watchpoints to HILTI to support that functionality. Only few scripts make use of that capability, most commonly for resolving DNS names.

```
global hosts: set[addr];

event connection_established(c: connection) {
    add hosts[c$id$resp_h]; # Record responder IP.
    }

event bro_done() {
    for ( i in hosts ) # Print all recorded IPs.
        print i;
}
```

(a) Bro event handlers in `track.bro`.

```
global ref<set<addr>> hosts = set<addr>()

[... Definitions for "connection" and "conn_id" ...]

hook void connection_established(ref<connection> c) {
  local addr __t1
  local ref<conn_id> __t2

  __t2 = struct.get c id
  __t1 = struct.get __t2 resp_h
  set.insert hosts __t1
}

hook void bro_done() {
    for ( i in hosts ) {
        call Hilti::print (i)
    }
}
```

(b) Compiled HILTI code (slightly simplified for readability).

```
# bro -b -r wikipedia.pcap compile_scripts=T track.bro
208.80.152.118
208.80.152.2
208.80.152.3
```

(c) Output with a small sample trace containing 3 servers.

Figure 8: Bro compiler example.

time interface to BinPAC++ parsers and (b) compiles Bro script code into HILTI and executes it. HILTI, BinPAC++, and the Bro plugin come with a total of about 850 unit tests ensuring correctness. Unless indicated otherwise, our implementation implements all functionality discussed in this paper, and we will release it to the community as open-source software under a BSD-style license. As a large part of the implementation represents an application of well-known compiler techniques, we focus on some of the more interesting aspects in the following.

Code Generation. The compiler `hiltic` receives HILTI machine code for compilation, as illustrated in Figure 2, which it then compiles into LLVM's instruction set. LLVM is an industrial-strength, open-source compiler toolchain that models a low-level but portable register machine. We also compile HILTI's runtime library into LLVM's representation, using the accompanying C compiler *clang*. We then link all of the parts together, first with a custom linker (see below), followed by LLVM's linker; and finally compile the result into native machine code. Besides generating machine code, LLVM also implements , domain-independent code optimizations. All parts of the code generation can either take place statically, producing a native binary to execute; or just-in-time inside a host application.

Linker. We add a specialized linker to our toolchain to support the HILTI compiler through custom transforma-

tions at the LLVM-level, enabling HILTI features that require a global view of all compilation units. For example, we maintain thread-local state per *virtual* thread and hence cannot leverage existing pthread-style functionality directly because it only separates the underlying hardware threads. HILTI's runtime associates with each virtual thread a separate array that contains a copy of all thread-local variables. However, as one compilation unit might access thread-locals defined in another, only the link stage has the global view necessary to determine the final layout of that array. Accordingly, our custom linker first merges all thread-locals across units into a single array structure, and then adapts all instructions to that aggregate layout. We use a similar link-time mechanism to support hooks across compilation units.

Runtime Model. With each virtual thread HILTI's runtime associates a context object that stores all its relevant state, including the array of thread-locals discussed above, the currently executing fiber (for suspending execution; see §3.2), timers scheduled for execution inside the thread, and exception status. Internally, we use a custom calling convention for compiled functions (as well as the runtime library) that passes the context object as an additional hidden object. HILTI currently propagates exceptions up the stack with explicit return values checks after each function call, which incurs a slight overhead. In the future we might switch to C++-style DWARF exceptions, which come with zero costs when not raising any exception.

HILTI garbage-collects dynamically allocated memory via reference counting—a technique Bro has proven a good match for the domain. While generally considered less efficient than mark-and-sweep collectors [24], reference counting is not only less complex to implement, but also—more importantly—its incremental operation and short deallocation times fit well with real-time constrains. The HILTI compiler emits reference count operations transparently during code generation as necessary. It does so rather freely at the moment, but we plan to add an optimization pass that removes redundant counter operations and also applies further optimizations similar to those that ARC [3] deploys. Our implementation currently lacks a cycle detector (as does Bro).

When implementing fibers, we considered several alternatives, including CPS-conversion [9] and manually copying the stack on demand. We settled on a `setcontext`-based scheme that proves both simple and efficient for supporting the large number of context switches that HILTI relies upon for implementing both incremental processing and virtual threads. The `setcontext` primitive allows a program to temporarily switch its stack over to a self-allocated memory segment, saving and later restoring processor registers to continue at the original location. A challenge with that approach lies in sizing the custom stacks correctly: if they remain too small, they will overflow; if too large, they will waste memory. While LLVM's code generator offers "segmented stacks" [11] as a mechanism to start with a small stack and then grow dynamically, this feature turns out com-

plex, non-portable, and rather inefficient. Instead we follow the lead of Rust [7] and leverage the MMU: we request large worst-case-sized memory segments with `mmap()` and rely upon the MMU to not actually allocate physical memory pages until it sees accesses. That approach performs well in practice: using libtask's optimized `setcontext` implementation [2] along with a custom free-list for recycling stacks, a micro-benchmark shows that the HILTI runtime can perform about 18 million context switches per second between existing fibers on a Xeon 5570 CPU. It furthermore supports about 5 million cycles per second of creating, starting, finishing, and deleting a fiber. We also confirmed that the memory usage indeed corresponds to the space in use, versus the amount allocated. We hence conclude that our fiber implementation provides the necessary performance.

Runtime Library. The runtime library implements the more complex HILTI data types—such as maps, regular expressions, and timers—as C functions called out to by the generated LLVM code. LLVM can inline function calls across link-time units and thus may eliminate the extra overhead of these invocations. The runtime library also implements the threading system with a corresponding scheduler, mapping virtual threads to native threads and scheduling jobs on a first-come, first-served basis. For functionality that requires serial execution, the runtime provides a command queue to send operations from arbitrary threads to a single dedicated manager thread. HILTI uses this mechanism, for example, for file output occurring from multiple threads concurrently. Generally, all runtime functionality is fully thread-safe, 64-bit code. However, we have not yet focused on efficiency when implementing runtime functionality. For example, we currently implement the `classifier` type as a linked list internally, which does not scale with larger numbers of rules. It will be straightforward to later transparently switch to a better data structure for packet classification [20].

Bro Interface. When adding the Bro interface for compiling BinPAC++ and scripts, we faced two main challenges: *(i)* we have to hook the generated code into Bro's processing at a number of points across its code base; and *(ii)* execution needs to translate back and forth between Bro's internal representation of data types and those of HILTI. For the former, we started with a Bro development version based on release 2.2 that comes with an early version of a new plugin interface, which enables us to load additional Bro functionality at runtime in the form of shared libraries. We extended that interface to provide further hooks into the script infrastructure and then implemented all HILTI-specific code as a plugin.

The translation between data types turned out to be more difficult. Internally, Bro represents all script values as instances of classes derived from a joint `Val` base class (e.g., there is an `EnumVal`, a `TableVal`, etc.). Unfortunately these instances are not only used by the script interpreter itself, but also passed around to most other parts of Bro's code base. Therefore, even when we replace the interpreter with our compiled code, to interface to the rest of Bro—e.g., for

feeding data into the logging system—we still need to create instances of these value classes. Similarly, we also need to convert Val's into the corresponding HILTI representation when generating events. Consequently, our HILTI plugin needs to generate a significant amount of glue code, which comes with a corresponding performance penalty as we discuss in §6. In practice, one would avoid this overhead by tieing the two systems together more directly .

6. EVALUATION

We now evaluate the HILTI environment by assessing the functionality and performance of the applications discussed in §4. We begin by summarizing our data and setup.

6.1 Data and Setup

To drive our applications with realistic workloads, we captured two full-payload packet traces at the border gateway of the UC Berkeley campus, exclusively containing HTTP and DNS traffic, respectively. The HTTP trace comprises a total of 30 GB of data in libpcap format, spanning 52 minutes of TCP port 80 traffic during a recent weekday morning and representing a sample of 1/25 of the link's total connections on port 80 during that time.[5] The trace contains about 340K HTTP requests/replies between 35K distinct pairs of hosts. The DNS trace comprises 1 GB in libpcap format, spanning about 10 minutes of UDP port 53 traffic during a recent weekday afternoon. Due to the lower overall volume of DNS, we could capture *all* of the campus' DNS during that interval. The trace contains about 65M DNS requests/replies between 435K distinct host pairs. We captured both traces with tcpdump, which reported no packet loss in either case. We chose the trace durations as a trade-off between including significant diversity and keeping their volume manageable for repeated offline measurements. We note that in particular for DNS, the number of requests/replies constitutes the primary driver for analysis performance, not raw packet volume. We conduct all measurements on a 64-bit Linux 3.12.8 system with two Intel Xeon 5570 CPUs, 24GB of RAM, and CPU frequency scaling disabled.

For the applications involving Bro we slightly adapt some of Bro's default scripts—which generate all of Bro's output files—by backporting a few recent bugfixes to the version we used. We also remove some minor dependencies on specifics of the built-in parsers. For DNS we furthermore disable per-request expiration timers, and for Bro's standard HTTP parser we disable recovering from content gaps; both features are not yet supported by our BinPAC++ versions. We use the modified Bro versions for all runs so that results are comparable across configurations. When measuring execution performance (in contrast to validating correct operation), we disable Bro's logging to avoid its I/O load affecting the results; Bro still performs the same computation but skips the final write operation. To measure CPU per-

---

[5]We captured the trace on a backend system behind a front-end load-balancer that splits up the total traffic on a per-flow basis [42].

formance, we instrument Bro to also record the time spent inside four different components of the code base: protocol analysis, script execution, HILTI-to-Bro glue code (see §5), and all remaining parts excluding initialization and finalization code.[6] We measure time in CPU cycles as retrieved via the PAPI library. Out of necessity these measurements remain somewhat approximate: *(i)* control flow within Bro is complex and one cannot always precisely attribute which subsystem to charge; and *(ii)* PAPI comes with overhead and fluctuations that affect the results [44]. However, after performing a number of cross-checks, we are confident that these measurements give us a solid qualitative understanding on how HILTI performs relative to standard Bro.

6.2 Berkeley Packet Filter

We first verify that running the compiled filter from Figure 4 on the HTTP trace performs correctly. Instead of the private addresses we use real ones from the trace that trigger the filter for about 2% of the packets. We link a basic libpcap-based driver program written in C with the compiled HILTI code. The driver reads the trace file inside a loop, calls the filter successively for each packet and increases a counter for every match. We then implement a second version of the driver that instead uses BPF to perform the same filtering, and we find that both applications indeed return the same number of matches. To understand the relative performance of the two filter implementations, we measure the CPU cycles spend inside the filtering code, finding that the HILTI version spends 1.70 times more cycles than BPF. A closer looks reveals that the HILTI-generated C stub is responsible for 20.6% of the difference, leaving an increase of 1.35 times when ignoring that; the runtime functionality that the stub facilitates remains unnecessary in this case, and the compiler could indeed just skip it. We deem the remaining difference acceptable, considering the HILTI's higher-level model compared to BPF's internal representation.

6.3 Stateful Firewall

We confirm the functionality of the stateful firewall application by comparing it with a simple Python script that implements the same functionality independently. We drive both instances with the DNS trace, configuring them with a small example rule set and feeding them with timestamp, source, and destination address for each packet, as extracted by `ipsumdump`. Both HILTI and Python versions parse the ipsumdump output into its components and then pass them as input into their rule matching logic. We confirm that the HILTI version produce the same number of matches vs. non-matches. It also performs the task orders of magnitude faster; however, given the slow nature of the Python interpreter, comparing times does not reveal much here.

6.4 Protocol Parsing

We next examine BinPAC++ protocol parsers, using Bro as the driver to feed them packet data. We take the HTTP and DNS parsers as case-studies and compare their results with Bro's standard, manually written C++ implementations.[7] We confirm correctness by running both BinPAC++ and standard parsers on the corresponding input trace and comparing the relevant Bro log files. For HTTP, `http.log` records all HTTP sessions along with extensive meta information such as requested URL, server response code, and MIME types of the message; and `files.log` records further information on the message bodies, including a SHA1 hash of their content. For DNS, `dns.log` records all DNS requests with queries, types, responses, and more.

Our BinPAC++ parsers attempt to mimic Bro's standard parsers as closely possible, however small discrepancies in analysis semantics remain hard to avoid for protocols as complex as HTTP and DNS. Before comparing the log files, we hence first normalize them to account for a number minor expected differences, including unique'ing them so that each entry appears only once.[8]

Table 2 summarizes the remaining differences. We find that for the normalized versions of `http.log`, 98.91% of all of the standard parser's log entries correspond to an identical instance in the BinPAC++ version; 98.36% for `files.log`. About half of the HTTP mismatches are due to a small number of "Partial Content" sessions, for which the BinPAC++ version often manages to extract more information. The remaining discrepancies reflect further semantic differences, often leading to different (or no) MIME types for an HTTP body. These mismatches then transfer over to the `files.log`, and we find neither parser consistently right in these cases. The (low) number of mismatches remains on the order of what we would expect for any two independent HTTP implementations. For `dns.log` we find almost perfect agreement, with >99.9% of the entries matching, and the remaining deviations being due to minor semantic differences (e.g., Bro's parser extracts only one entry from TXT records, BinPAC++ all; the BinPAC++ parser does not abort as easily for traffic on port 53 that is not in fact DNS). Overall, we conclude that the BinPAC++ results closely match those of Bro's standard parsers, which means we can proceed to meaningfully compare their performance, as they indeed perform essentially the same work.

Figure 9 summarizes CPU times. The plot breaks down the time spent inside Bro components when using its standard parsers vs. the HILTI-based versions. The time for "Protocol Parsing" constitutes the key metric: the Bin-

---

[6]This also excludes compiling HILTI code at startup. While that can take noticeable time, one could cache the machine-code for reuse on later invocations.

[7]Note that we indeed compare against manually written C++ implementations, *not* code generated by the classic BinPAC discussed in [34]. While Bro uses BinPAC for some of its protocols, it does not do so for HTTP or DNS.

[8]The normalization further includes adjustments for slight timing and ordering differences, a few fields with size information that the BinPAC++ parsers cannot easily pass on to Bro for technical reasons, and formatting differences in field content.

#Lines	http.log		files.log		dns.log	
	Std	Pac	Std	Pac	Std	Pac
Total	338K	338K	273K	273K	2573K	2573K
Normalized	338K	338K	273K	273K	2492K	2492K
Identical	98.91%		98.36%		>99.9%	

Table 2: Agreement HILTI (Pac) vs. standard (Std) parsers.

#Lines	http.log		files.log		dns.log	
	Std	Hlt	Std	Hlt	Std	Hlt
Total	338K	338K	273K	273K	2573K	2573K
Normalized	338K	338K	273K	273K	2492K	2492K
Identical	>99.99%		99.98%		>99.99%	

Table 3: Output of compiled scripts (Hlt) vs standard (Std).

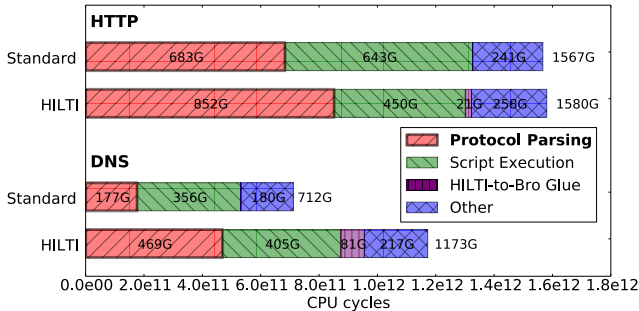Figure 9: Performance of HILTI-based protocol parsers.

Figure 10: Performance of scripts compiled into HILTI.

PAC++ parsers need 1.28 times and 3.03 times more CPU cycles, respectively, when running on the HTTP and DNS traces than the standard implementations. For HTTP that means HILTI already comes close to the performance of manually written C++ code. For DNS, the slowdown is more significant, though we argue still acceptable given the current prototype state of our compiler implementation with its potential for further optimization and tuning. We profiled the DNS analysis in more detail on a trace excerpt and found two particular opportunities for improvement. First, BinPAC++ parsers perform more memory allocations, and the effect is particularly pronounced for DNS: when using the Bin-PAC++ parser Bro performs about 47% more memory allocations (vs. 19% more for HTTP). The increase comes from frequent instantiation of dynamic objects during the parsing process—likely a similar overhead as classic BinPAC exhibits as well, and with similar solutions [39]. Second, currently the BinPAC++ compiler always generates code supporting incremental parsing, even though it could optimize for UDP where one sees complete PDUs at a time (as Bro's standard parser indeed does).

Figure 9 also shows the time spent inside the HILTI-to-Bro data conversion glue: 1.3%/6.9% of the total cycles, respectively—an overhead that would disappear if HILTI were more tightly integrated into the host application. Interestingly, we also see in Figure 9 that for HTTP, Bro spends less time in script execution when using the BinPAC++ analyzer, even though it follows the same code path. We tracked down this difference to the standard parser generating a large number of redundant events related to file analysis, which the BinPAC++ parser omits (about 30% more in total). While the impact of these events on parsing and output remains negligible, they lead to more load on the script interpreter. Recent Bro versions have fixed this problem.

6.5 Bro Script Compiler

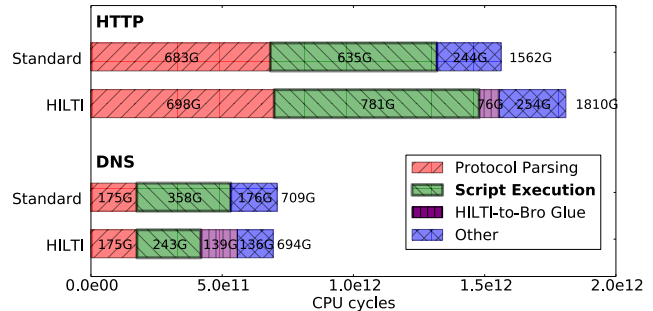Turning to the script compiler application, we again first verify its correct operation. Using the same methodology

as with the protocol parsers, we compare the log files that the compiled HILTI versions of the HTTP and DNS scripts produce with the output of Bro's standard script interpreter. Table 2 summarizes the differences running on the corresponding traces. We see an excellent fit in all three cases. Inspecting the few cases where the two versions do not agree, we find that for http.log and files.log almost all differences are due to fully insignificant output ordering when logging sets of values—which our normalization cannot not easily account for. For dns.log the only differences come from an IPv6 address logged in a slightly different output format. Overall, we conclude that the compiled HILTI code produces the same output as Bro's standard interpreter, and we thus confirm that HILTI's model can indeed capture a complex domain-specific language.

Next we turn to execution performance. As a simple baseline benchmark, we first execute a small Bro script that computes Fibonacci numbers recursively. The compiled HILTI version solves this task orders of magnitude faster than Bro's standard interpreter, which demonstrates the potential for compiling scripts into machine code. However, this example represents the best case for HILTI: it requires hardly any interaction between HILTI and Bro's C++ core, and it allows LLVM's code generation to shine by producing optimal machine code. As more realistic examples, Figure 10 compares execution performance with the HTTP and DNS scripts, respectively. For HTTP, the HILTI version remains slightly slower, requiring 1.30 times the cycles. For DNS, HILTI proves 6.9% faster. As with the protocol parsers, the glue code adds further overhead—4.2% and 20.0% of total cycles, respectively—which a fully integrated host application would not incur. Overall we conclude that the compiled HILTI scripts exhibit performance similar to Bro's existing scripting system. One could argue that compiled scripts should decrease execution times more, compared to Bro's standard interpretation. While indeed we expect that further improvements of our toolchain will achieve that, we also point out that it remains challenging to quantify the po-

tential: with a high-level scripting language, operations on data structures (e.g., hash maps) and other runtime library functionality, including dynamic memory management, account for a significant share of the work load that can only improve to a certain extent. Furthermore, even when interpreted, Bro's statically typed language can execute much faster than dynamically typed environments such as Python or Perl, which would not be feasible to use for real-time network traffic analysis in the first place.

6.6 Summary

We have shown that *(i)* HILTI correctly captures the semantics of all four application scenarios, from low-level protocol parsing to complex high-level analyses; and *(ii)* its runtime performance generally aligns with that of the existing systems. While sometimes slower, it remains on the order of what we expect for the current state of the implementation, i.e., a 4-stage compilation process (host application, HILTI, LLVM, machine code) only partially optimized (LLVM) versus manually written production code. Indeed, our toolchain does yet exploit HILTI's optimization potential: it lacks support for even the most basic compiler optimizations, such as constant folding and common subexpression elimination at the HILTI level (e.g., while LLVM cannot identify subsequent lookups for the same map element, these would be easy to compress inside HILTI).

We have not yet pursued parallelizing applications inside HILTI, as that would involve a number of further aspects in terms of analysis structure and evaluation, exceeding the scope of this paper. However, we have verified HILTI's thread-safety guarantees, as well as correct operation of the scheduler, by load-balancing DNS traffic across varying numbers of hardware threads, each processing their input with the corresponding HILTI-based parser. As expected, we found the same HILTI parsing code to support both the threaded and non-threaded setups.

7. DISCUSSION

Safe Execution Environment. As a key design aspect, HILTI provides a safe runtime execution environment that prevents unintended data and control flows when facing unexpected input. For example, HILTI's instructions generally validate their operands to avoid undefined behavior, such as out-of-bounds iterators; and the memory management prevents dangling references. Furthermore, by separating state between threads, HILTI provides a well-defined setting for concurrent analysis: HILTI code is always safe to execute in parallel. While some of these safety properties come with performance penalties, we deem them well worth their cost, compared to the traditional alternative of writing C code that parses untrusted input, potentially concurrently, in real-time.

Performance via Abstraction. We also designed HILTI's machine model to preserve high-level domain semantics that offer extensive potential for global, transparent optimization. While not yet a focus of our prototype implementation, we

expect HILTI to eventually facilitate a range of powerful code enhancements. Consider our two Bro-based applications, protocol parsing and script analysis: taken together, they effectively port the *main part of Bro* over to the HILTI platform. While today parsing and analysis remain separate Bro subsystems, HILTI allows us to reason about them simultaneously inside a single model, with the opportunity to then optimize functionality across their boundaries. For example, if a user does not configure any Bro scripts that inspect bodies of HTTP messages, HILTI could simply remove the code for parsing them in depth.

Generally we see opportunities for automatic optimization in four main areas. First, HILTI facilitates *improving individual functionality* by tuning the implementation. As a simple example, we plan to better adapt HILTI's `map` type to real-time usage by avoiding CPU spikes during hash table resizes [14]. Likewise, we are considering JIT compilation of regular expressions, similar to `re2c` [4]. HILTI allows to perform such fine-tuning "under the hood" while transparently benefiting any host application.

Second, HILTI's abstractions enable *transparent integration of non-standard hardware capabilities*. Traditionally, integrating custom hardware elements (e.g., FPGA pattern matchers, dedicated lookup modules, fast packet classification) into a networking application requires significant effort to manually adapt the code. HILTI, however, can switch to a different code path as suitable, either at compile-time or dynamically during execution, without changing the interface to the host application. For example, preliminary simulations show that HILTI's hash tables map well to PLUG [12].

Third, HILTI's execution model facilities *compiler-level code optimization* by providing context for control and dataflow analyses [25]. For example, state management can benefit from grouping memory operations for improved cache locality, according to access patterns that the HILTI representation suggests (e.g., containers could structure memory operations with element expiration times in mind). Also, optimizations can help remove overhead coming with HILTI's runtime model, such as skipping unnecessary exception checks, moving bounds-checking from runtime to compile time where statically safe, and optimizing memory management and garbage collection (e.g., BinPAC++-style parsers could often move intermediary heap objects to less expensive stack storage). Another powerful technique concerns elimination of unneeded code at link-time, as in the Bro example we sketch above: the HILTI linker can remove any code (as well as state) that it can statically determine as unreachable with the host application's parameterization.

Fourth, we are confident that HILTI has sufficient context available to *automatically infer suitable parallelization strategies* for many networking applications. By analysing data flows, potentially augmented with feedback from runtime profiling, it can leverage the typical unit-based structure of network analysis tasks for scheduling them scalably across CPUs, as outlined in [41].

8. RELATED WORK

By their very nature, existing abstract machine implementations focus on specifics of their respective target domains, and to our knowledge none fit well to the requirements of high-performance, flexible network traffic analysis. This includes machine models underlying typical programming languages (e.g., JVM, Parrot VM, Lua VM, Erlang's BEAM/HiPE [37]). Despite raising the level of abstraction, these machines do not offer high-level primitives to efficiently express problems of the domain. 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 provides 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 [1] implements basic building blocks commonly used by NIDS's, paying particular attention to a design robust when facing adversaries and evasion [22]. 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 [29] 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 [26] allows users to compose a software router from elements that encapsulate predefined primitives, such as IP header extractors, queues, and packet schedulers. Rather than mapping the custom configuration language to the underlying C++ engine, Click configurations could alternatively compile into HILTI. RouteBricks [16] is a multi-Gbps distributed software router. To optimize intra-node parallelization, Route-Bricks uses techniques akin to HILTI's concurrency model: per-core packet queues enable an efficient lock-free programming model with good cache performance. HILTI can easily express such per-flow analysis (within a single system) by routing related analysis to the same thread, and its threading model allows for other scheduling strategies as well. NetPDL [38] 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 would nicely map to HILTI. Xplico [6] is a network forensic tool written in C that ships with protocol analyzers and manipulators.The HTTP analyzer, for example, reassembles HTTP payload by writing the packet contents into per-flow files on disk, which higher-level analyzers (such as webchat) then re-read for further analysis. HILTI's incremental and suspendable stream parsing 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 [31] provides a vendor-agnostic interface to such functionality, and a number of higher-level languages [19, 27, 17, 32] use it to control compatible hardware devices. By adding an OpenFlow interface to HILTI, it could become a corresponding backend to drive the software component of such systems and dynamically control network policies based on traffic analysis. NetVM [33] 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 has a lower-level focus than HILTI because it primarily attempts to achieve portability of signature matching. Contrary to the VM isolation model of NetVM, HILTI's compilation into LLVM code enables late and global optimizations, whereas it appears difficult to optimize across NetVM elements. Similar to our example in §4, Linux has added support for JIT compiling BPF expressions into native assembly [10]. FreeBSD 7 also includes experimental BPF JIT support. 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., [40, 18]), parallelization on GPUs (e.g., [43, 8, 21]), robust TCP stream reassembly [13], and high-speed lookup tables such as PLUG [12]. HILTI's design allows to transparently offload specific computations to such specialized hardware when available.

9. CONCLUSION

We present the design and implementation of HILTI, a platform for deep, stateful network traffic analysis. HILTI represents a middle-layer located between a host application and the hardware platform that executes the analysis. We argue that while networking applications tend to share a large set of common functionality, they typically reimplement it from scratch each time, raising the possibility of falling into pitfalls that others have previously mastered. HILTI bridges that gap by providing a common substrate to applications to support their implementation while facilitating reuse of high-level components built on top of the platform. We developed a prototype compiler that implements all of HILTI's functionality, including rich domain-specific data types, automatic memory management, flexible control flow, concurrent execution, profiling and debugging support, and an extensive API for host applications. We further built four example applications on top of HILTI that demonstrate its ability to support a range of typical network analyses. We plan to advance HILTI further into a platform suitable for operational deployment in large-scale network environments by

exploiting its full performance potential through transparent optimization and integration of non-standard hardware elements. We also envision HILTI to become a platform for networking research by facilitating rapid prototyping of novel network functionality.

10. REFERENCES

[1] Libnids. http://libnids.sourceforge.net.
[2] libtask. http://swtch.com/libtask.
[3] Objective-C Automatic Reference Counting (ARC). http://clang.llvm.org/docs/AutomaticReferenceCounting.html.
[4] re2c. http://re2c.org.
[5] Suricata Source Code - src/flow-hash.c. https://github.com/inliniac/suricata/blob/master/src/flow-hash.c.
[6] Xplico. http://www.xplico.org.
[7] B. Anderson. Abandoning segmented stacks in Rust. https://mail.mozilla.org/pipermail/rust-dev/2013-November/006314.html.
[8] 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.
[9] A. W. Appel. *Compiling with Continuations*. CUP, 1992.
[10] J. Corbet. A JIT for packet filters. http://lwn.net/Articles/437981/.
[11] S. Das. Segmented Stacks in LLVM. http://www.google-melange.com/gsoc/project/google/gsoc2011/sanjoyd/13001.
[12] 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, 2009.
[13] S. Dharmapurikar and V. Paxson. Robust TCP Stream Reassembly in the Presence of Adversaries. In *USENIX Security*, 2005.
[14] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Operational Experiences with High-Volume Network Intrusion Detection. In *ACM CCS*, Oct. 2004.
[15] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the Resource Consumption of Network Intrusion Detection Systems. In *Proc. RAID*, 2008.
[16] 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.
[17] N. Foster et al. Frenetic: A High-Level Language for OpenFlow Networks. In *Proc. PRESTO*, 2010.
[18] R. Franklin, D. Carver, and B. Hutchings. Assisting Network Intrusion Detection with Reconfigurable Hardware. In *Proc. FCCM*, 2002.
[19] 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, 2008.
[20] P. Gupta and N. McKeown. Algorithms for Packet Classification. http://yuba.stanford.edu/~nickm/papers/classification_tutorial_01.pdf, 2001.
[21] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-accelerated Software Router. In *Proc. ACM SIGCOMM*, 2010.
[22] M. Handley, C. Kreibich, and V. Paxson. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proc. USENIX Security*, 2001.
[23] S. Ioannidis, K. Anagnostakis, J. Ioannidis, and A. Keromytis. xPF: Packet Filtering for Lowcost Network Monitoring. In *Proc. IEEE HPSR*, pages 121–126, 2002.
[24] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Cambridge University Press, 2011.
[25] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002.
[26] 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.
[27] T. Koponen et al. Onix: A Distributed Control Platform for Large-Scale Production Networks. In *USENIX OSDI*, 2010.
[28] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Symposium on Code Generation and Optimization*, 2004.
[29] Z. Li et al. NetShield: Massive Semantics-Based Vulnerability Signature Matching for High-Speed Networks. In *Proc. ACM SIGCOMM*, 2010.
[30] 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.
[31] N. McKeown et al. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38:69–74, 2008.
[32] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. In *Proc. POPL*, 2012.
[33] O. Morandi, G. Moscardi, and F. Risso. An Intrusion Detection Sensor for the NetVM Virtual Processor. In *Proc. ICOIN*, 2009.
[34] R. Pang, V. Paxson, R. Sommer, and L. Peterson. binpac: A yacc for Writing Application Protocol Parsers. In *ACM Internet Measurement Conference*, 2006.
[35] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24), 1999.
[36] 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 *Proc. USENIX Hot Security Workshop*, August 2006.
[37] M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang Compiler: System Description and Performance Evaluation. In *Proc. FLOPS*, 2002.
[38] F. Risso and M. Baldi. NetPDL: An Extensible XML-based Language for Packet Header Description. *Computer Networks*, 50:688–706, April 2006.
[39] N. Schear, D. Albrecht, and N. Borisov. High-Speed Matching of Vulnerability Signatures. In *Proc. RAID*, 2008.
[40] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *Proc. IEEE FCCM*, Apr. 2001.
[41] 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.
[42] 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.
[43] 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.
[44] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of Performance Counter Measurements. In *IEEE ISPASS*, 2009.