HILTI: An Abstract Execution Environment for High-Performance Network Traffic Analysis

Robin Sommer, Nick Weaver, and Vern Paxson

TR-10-003

February 2010

Abstract

When building applications that process large volumes of network traffic—such as high-performance firewalls 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 that provides a bridge between these two levels by offering to the application designer the high-level abstractions required for effectively describing typical network analysis tasks, while still ensuring the performance necessary for monitoring Gbps networks in operational settings. This new middle-layer comprises 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 highly optimized, natively executable task-parallel code for a given target platform. We present the design and an early prototype of the new environment and discuss opportunities for extensive compile-time code optimizations that our approach enables by leveraging domain-specific context. Such an environment holds promise for unleashing the community's potential to build libraries of efficient analysis functionality, reusable across a wide range of scenarios.

This work was made possible by National Science Foundation grants NSF-0831535 ("Comprehensive Applications Analysis and Control") and NSF-0915667 ("A High-Performance Abstract Machine for Network Intrusion Detection").

1 Introduction

When building applications that process large volumes of network traffic—such as high-performance firewalls or intrusion detection systems (IDS)—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. As a networking application is assembling the network's high-level picture from zillions of individual packets, it must not only operate extremely efficiently to achieve line-rate performance under real-time constraints, but also deal securely with a stream of untrusted input that requires conservative processing. However, despite this challenge, there is hardly any reuse of existing, well-proven functionality across applications: implementations tend to build the same type of components from scratch, often missing out on opportunities to leverage experience from existing deployments.

We believe that much of this can be explained by the lack of a common platform that provides high-level abstractions suitable to implement typical network analysis tasks in a reusable fashion, yet still offers the flexibility and performance that a manually written low-level implementation would yield. In this work, we develop such a platform. We present a new middle-layer for network traffic processing, consisting of two main pieces: *(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 highly optimized, natively executable code for a given target platform, with performance comparable to manually written C code. At the core of the abstract machine model is a *high-level intermediary language for traffic analysis (HILTI)*. HILTI provides high-level data structures, powerful control flow primitives, extensive concurrency support, and a secure memory model along with protection from unintended control and data flows. A corresponding compiler turns HILTI programs into input for the open-source *Low-Level Virtual Machine (LLVM)* infrastructure, which we leverage for all target-specific generation of native code.

The broader goal of our undertaking is to provide the networking community with a novel architecture that facilitates development and reuse of building blocks commonly required for network traffic analysis. While the focus of our effort is the design and implementation of the HILTI environment itself, we envision enabling the community to build analysis functionality on top of the new platform, eventually developing a library of reusable high-performance analysis functionality.

We present the HILTI model in more detail in § 2, including an early prototype that already provides many of HILTI's key features. In § 3, we discuss application scenarios leveraging the new platform, and we then examine the key challenge for achieving high performance—domain-specific code optimization—in § 4. We finish with a discussion of design choices in § 5.

2 The HILTI Environment

Our overall goal is to compile high-level descriptions of network analyses into efficient native code that can then be executed directly on a specific target platform. To this end, the heart of our effort is the design and implementation of an *intermediary layer* within this process: an abstract machine environment that *(i)* provides abstractions to an application suited to express typical analysis tasks; and *(ii)* compiles analysis expressed in such abstract terms into native, high-performance code.

Figure 1 summarizes the design of the HILTI environment. The workflow starts from an *analysis specification*, expressed in an application-specific input format. For example, for a firewall this could be the list of rules; for an IDS its signature set. An application-specific *analysis compiler* transforms such a specification into HILTI's high-level instruction set. This instruction set is the conceptual centerpiece of HILTI's *execution model*, which also includes corresponding execution semantics, a *HILTI compiler*, a *stub generator* for interfacing the *host application* via language-independent *interface descriptions*, and a *runtime support library*. The compiler turns HILTI code into the instruction set of the *Low-Level Virtual Machine (LLVM)* [5] for low-level code generation. Finally, the byte-code is linked with the host application into a native executable.

In the following, we present our main focus areas in § 2.1, and then discuss the execution model in § 2.2 and our current prototype in § 2.3.

2.1 Focus Areas

HILTI needs to balance between providing a flexible, generic platform for a variety of network applications, and producing efficient and robust code at the end of the processing pipeline. We now discuss the main areas we need to consider when designing the framework.

Instruction set. The high-level instruction set needs to support typical analysis idioms directly. In particular, much abstraction can be achieved by providing rich domain-specific data types, automatic state management, and a flexible control flow model, in particular allowing for both *concurrent* and *asynchronous* code execution.

Robust execution. A crucial concern is providing a secure run-time environment for the generated native

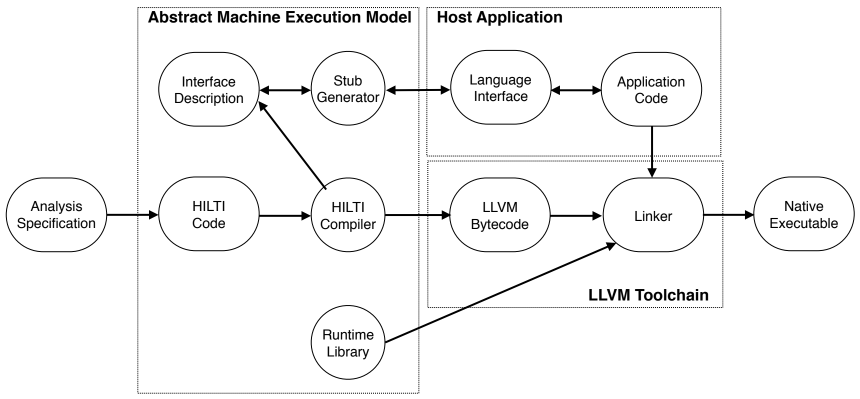

Figure 1: Architecture of the HILTI environment.

code, in which neither programming errors nor unexpected input traffic can lead to malicious control- or data-flows. Specifically, we require the HILTI language to be statically type-safe and provide mechanisms to handle unexpected situations robustly.

Compile-time optimization,. The key to achieving efficient output is extensive code optimization. The HILTI-level provides the kind of domain-specific context that allows modern compiler techniques to excel [4]. To support such an approach, the execution model should make relevant semantics explicit (e.g., processing structure, data flows, memory operations, operand types) and limit variability by restricting the number of ways idioms can be expressed.

Host application interface. The host application drives the analysis and must therefore be able to control HILTI's operation as well as hook into its processing as necessary for retrieving results. However, it is crucial to account for the fact that applications can internally be structured in very different ways.

2.2 Execution Model

At HILTI's core is an instruction set that models a high-level domain-specific register machine. We now discuss the instruction set along with its corresponding execution semantics in terms of syntax, data types, memory management and concurrency support.

Syntax. We model the HILTI language after register-based assembler languages. A program consists of a series of instructions of the general form `<target> = <instruction> <op1> <op2>`, with target/operands omitted where not needed. In addition, there are primitives to define functions; custom data types (including tuples and structures); local and thread-global variables; and exceptions. Instruction mnemonics are of the form `<prefix>.<operation>`, with the same `prefix` indicating a joint set of functionality. In particular, for data types we use a convention in which the `prefix` refers to the type and the first operand is the object to be manipulated (e.g., `list.append mylist 42`, which appends the integer 42 to the list specified as the operand, `mylist`). Generally, the instruction set deliberately limits syntactic flexibility, supporting compiler transformations directly without needing another intermediary representation. Figure 2 shows an example program.

Data Types. While being parsimonious with syntax, we provide HILTI with a rich set of data types while enforcing static safety. First, in addition to standard atomic types such as integers, character sequences (with separate types for Unicode strings and raw bytes), floating points, and enums, it offers domain-specific primitives such as *time intervals*, *network addresses*[1], *subnet masks*, *ports*, and *regular expressions*. While these are technically straight-forward to implement, they provide crucial context for type checking as well as for data flow and dependency analyses.

Second, HILTI provides a set of high-level composite types, including lists, vectors, sets, maps, stacks, and queues. In our experience, typical analysis tasks rely heavily on such data structures and having them directly available enables expressing an analysis in very natural terms. For type-safety, all containers are statically parametrized. They also provide built-in state management support, such as automatic expiration of elements after a specified amount of time or when reaching an upper size limit. Further domain-specific types are *overlays* for safely and efficiently dissecting packet headers into their components; and "channels" for transferring large volumes of (typed) data either internally or between host applications and the HILTI environment, such as a stream of network packets or the payload of a TCP session. Channels are also one of HILTI's main primitives for supporting concurrency transparently (see below).

[1]The data type transparently supports both IPv4 and IPv6 addresses.

```
typedef tuple<addr, addr> host_pair         # Define custom tuple type.
global ref< set<host_pair> > attempts       # Define two global references.
global ref< map<addr, int32> > counters

# Initialization code to be called at startup.
  [...]
  attempts = new set<host_pair>             # Allocate set on the heap.
  set.expire attempts 300 decr_count        # Expire entries after 300s; decr_count() will be
                                            # called to decrease attempt counter (not shown).
  counters = new map<addr, int32>           # Allocate map on the heap.
  map.set_default counters 0                # Non-existing entries default to 0.
  [...]

# Function to be called for each attempted connection.
void connection_attempt(addr src, addr dst) {
    local bool cond                         # Define local variables on the stack.
    local int32 count
    local string message

    cond = set.contains attempts (src, dst) # Check whether set contains tuple.
    jump.if cond exit                       # If yes, pair is already known.
    set.insert attempts (src, dst)          # Insert tuple into set.

    count = map.lookup counters src         # Lookup attempt counter (default 0).
    count = int.add count 1                 # Increase counter.
    map.insert counters (src, count)        # Store new value.

    cond = int.lower count 20               # Check if threshold of 20 reached.
    jump.if exit                            # It is not, so we are done.

                                            # Build alarm message printf-style.
    message = call Hilti::fmt ("%s has reached threshold", (src))
    call alarm (message)                    # Call alarm function to report.
exit:
   return                                   # Return from function.
}
```

Figure 2: Example of possible code implementing a simple detector for address scans. `connection_attempt` will be called for each attempted connection, either by another component or by the host application. It counts the number of attempts to unique destinations per source over a sliding 300 s interval. Note that the model for such code is to be auto-generated.

Finally, HILTI contains first-order support for both *closures* and *one-shot continuations*. These two building blocks enable it to provide a set of types enabling *asynchronous execution* of code blocks, including *timers* for scheduling execution to a specific time in the future; *snapshots* taken of the current execution state for later resumption; *triggers* for execution of code when a specified condition occurs; and *hooks* for optional external code to run at specific points during a computation.

Memory Model. The main objectives for HILTI's memory model are type-safety and automatic memory management. All operands are strictly typed, with no implicit casts, so that the compiler can enforce restrictions. An explicit `new` instruction makes allocations of dynamic memory explicit. Its return type is a *reference* type `ref<T>`, with `T` indicating the type of the allocated object. Dynamically allocated memory is managed by the HILTI runtime and eventually garbage-collected.

Concurrency Model. A crucial capability to provide is concurrent processing. Most directly, this refers to "real" concurrency in terms of having the ability to use multiple threads for parallel execution of HILTI code. With novel many-core platforms emerging rapidly, such a model is important to support effectively. Perhaps less obvious, there is a second form of concurrency inherent to how network analysis operates: tasks tend to be structured in terms of certain *processing units*. For example, many IDS rely on *flows* as their primary unit and express their detection logic as a series of steps working on a single flow at a time (e.g., "alarm when an HTTP flow requests a particular URL"). However, while such a policy is conceptually expressed at the level of an individual unit, at runtime the system needs to *multiplex* the analysis across a large number of simultaneous units.

To support both of these types of concurrency, HILTI implements two orthogonal concurrency mechanisms. First, HILTI directly supports concurrent operation when running on multi-core platforms by providing a thread abstraction, thread-local storage, and a set of atomic memory operations. Second, independent of any actual parallel execution, HILTI provides extremely lightweight *virtual threads* across which it can transparently multiplex processing using using continuation-based [3] cooperative multitasking. Now, these two concurrency models can be combined to provide the host application with an abstract view of having a infinite supply of extremely lightweight threads at its disposal, which are then at runtime automatically mapped to a much smaller set of actual hardware threads.

2.2.1 Compiler

The HILTI compiler turns code expressed in the high-level instruction set into LLVM bytecode, which is then further compiled into native code. Generally, most of the compilation process is straight-forward to implement using standard compiler technology. In the following, we highlight a few of the more interesting issues.

To support continuations, the HILTI compiler generates code that manages custom stack frames explicitly, relying on garbage collection to determine when a frame is safe to release.

For some functionality, the generated code relies on support by a runtime library linked to the final native code. One example is some of the more complex data types, such as hash maps: operations like inserting an element into a map are outsourced to the runtime library.

The compiler needs to enable the host application to interface with the generated code in a flexible way. We provision for this with two separate mechanisms. First, most communication uses basic function calls: the host application can call functions defined in HILTI code, and vice versa. While technically a bit subtle—we need to convert data types and adapt calling conventions—the compiler generates most of the glue logic automatically. Due to this glue, such function calls however incur a small overhead. When passing larger volumes of data, the channel data type (see above) is more efficient as internally it avoids copying the data physically. Accordingly, we provide a channel API to the host application.

2.2.2 Linking and Execution

A host application can use the code generated from a HILTI program in two different ways. First, it can statically link the code into the application's core, producing a single executable. This model will best suit cases where the analysis rarely changes, requiring new executables only occasionally. The second option is to load the generated code at *runtime*, either as a dynamic library or by leveraging LLVM's just-in-time compiler. The latter approach provides two additional, powerful capabilities: *(i)* the portable nature of LLVM's bytecode allows distributing a compiled analysis to other parties (e.g., an IDS could receive detector updates in this form); and *(ii)* it makes it easy to *replace* analysis during runtime without the need for a restart of the host application (e.g, the IDS could pick up a new signature set, compile it, and switch processing over).

2.3 Current Prototype

We have implemented an initial prototype of HILTI's main pieces. The HILTI compiler is written in Python and the run-time library in C. Many features discussed above are already working, including many data types (such as IPv4 and IPv6 addresses, ports, lists, vectors, regular expressions, overlays, channels); as well as continuations, exceptions, threads, and the generation of glue logic for communication between HILTI and host applications. The generated code is not yet particularly optimized for efficiency but we expect to spend significant time on improving performance once the overall framework gets into a stable state. As a first proof-of-concept host application, we implemented a basic compiler for tcpdump-style BPF filters [6], producing native code that filters network packets according to a specified expression. Figure § 3 shows an example. The code leverages an IP "overlay" type provided by HILTI for transparent, type-safe access to individual header fields.

```
bool filter(ref<bytes> packet) {
    [... Declarations of t1,...,t5 ... ]
    local IP::Header iphdr
    local iterator<bytes> start

    # Attach IP header overlay to packet's raw bytes.
    start = bytes.begin packet
    overlay.attach iphdr start

    # Extract fields.
    t1 = overlay.get iphdr "src"
    t2 = equal t1 192.168.1.1
    t3 = overlay.get iphdr "dst"
    t4 = equal t3 192.168.1.2
    t5 = bool.or t2 t4
    return.result t5
    }
```

Figure 3: HILTI code for the filter `‘‘src host 192.168.1.1 or dst host 192.168.1.2’’`.

3 Application Scenarios

A wide variety of host applications can leverage the functionality provided by the HILTI environment, including firewalls, security monitors, application-layer proxies, and traffic debuggers. For example, by compiling its rule set into HILTI code, a firewall system can avoid the low-level complexity of parsing packets and managing its own data structures. Likewise, intrusion detection systems can specify their detection logic on top of HILTI-provided primitives. Consider for example the popular open-source Snort IDS. Snort could not only transform its signatures into a HILTI module, but also leverage the platform's capabilities for supporting its special-purpose detectors currently written in low-level C code. Another open-source IDS, Bro, provides a custom scripting language that is currently *interpreted* at run-time, rendering it a major performance bottleneck. Compiling Bro scripts into HILTI code, and from there into a native executable, would boost the system's run-time performance.

From a different perspective, HILTI allows applications to *share* common functionality easily. HILTI enables writing code once on top of its execution model and then integrates it transparently into different host applications. As a specific example for sharing functionality, consider parsers for network protocols as found in almost any traffic analysis application. A key step towards their reuse is the BinPAC [7] parser generator—"a yacc for network protocols"—which generates parser implementations from declarative grammar specifications. Currently, BinPAC focuses on parsing *syntax*, still leaving tracking of communication *semantics* to the main application. By generating code for the HILTI platform (rather than C++), and extending it with corresponding *semantic* constructs, BinPAC could generate fully *standalone* parsers suitable for use in many applications.

One little explored area of working with network traffic is the challenge of *understanding* and potentially *restricting* an analysis. Due to privacy implications, inspecting real-world network traffic is a highly sensitive task. Consequently, operators often face the challenge of assessing a proposed analysis for its potential to leak information. As an example from the research world, consider the scientist asking to analyze traffic on its behalf in a mediated setting. Clearly, the operator must fully understand the nature of any results handed back. By capturing the analysis at a high semantic level, the HILTI environment can *(i)* provide support for tracking information flows, e.g., with taint analysis; and *(ii) enforce* constraints during run-time by restricting permitted operations, turning HILTI into a domain-specific sandbox.

Finally, we note that every application using the HILTI environment benefits directly from any additional resources the framework can leverage, such as specific hardware support for any of its functionality. Consider for example a platform providing an FPGA-based pattern matcher: once we enable the HILTI runtime to interface with it, all string matching operations will benefit with no further change to any host application.

4 Performance via Optimization

Code optimization is the key to providing an abstract machine environment that is sufficiently generic to cater to different host applications, yet can still exploit their full performance potential individually. Our primary focus is *domain-specific* optimizations at the interface between HILTI and LLVM, where we find a large potential for optimizing both CPU and memory performance. Likely the most important single area here is the concurrency model: we can achieve significant performance gains by adapting control flow and memory access patterns to the highly parallel analysis structure inherent to the analysis [4]. As one example, many applications analyze connections mostly independent of others (e.g., when decoding protocols), and we can schedule the processing across a set of concurrent threads automatically. Generally, in network traffic analysis, there is a wealth of such inherent task parallelism [8], and HILTI's domain-specific semantics provide parallelization techniques with crucial context.

A further area for high-level optimizations is state management. As HILTI's data types have state-management support built-in, much of the relevant information is readily available for understanding their concrete semantics (such as where and when entry expiration takes place). This allows for a set of powerful static optimizations, such as grouping related state management tasks for joint execution, and it also provides the opportunity for optimization based on *runtime profiling*. Strategies developed in the past for *automatic tuning* (e.g., [2]) should also transfer well to the HILTI environment, in particular because the execution model's reduced structural complexity (compared to the full IDS used in the cited study) will make resource estimates more precise.

Generally, for all low-level, domain-independent optimizations we rely on the LLVM infrastructure, which implements many standard compiler techniques that are crucial for generating efficient code. However, relying on LLVM's optimizations also enables us to provide a generic interface for host applications without sacrificing performance. Consider HILTI code that wants to report different kinds of information back to an application depending on its requirements. Some of the information might be expensive to compute and thus the corresponding code should better be excluded if the data is not going to be needed. LLVM in fact enables such scenarios by performing optimizations across link-time units, in particular recursive dead code elimination. This allows the HILTI compiler to initially produce "worst-case code" by including all functionality that could *potentially* be used.

5 Discussion

Building an execution environment. One might wonder about our choice to build a complete abstract execution environment rather than assemble reusable functionality into, say, a simple C library. The key here is the additional power that the compilation of HILTI code provides. Typical traffic analysis logic is built on top of 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. Furthermore, compilation also enables us to provide safety guarantees that a C library cannot achieve.

Leveraging existing technology. Many abstract machines (also called virtual machines) have been built for a variety of target domains, with the the Java Virtual Machine likely being the best known instance. Dynamically typed languages (e.g., Python, Ruby) often rely on them for efficient run-time operation. The Parrot VM [1], originating in the Perl community, aims to provide a common platform for such languages, similar in spirit to what we propose to build for networking applications. Abstract machines are by their very nature specific to their particular domain, and it is therefore not surprising to find none of the existing ones fit well with our focus areas. However, these mismatches concern primarily high-level, domain-specific functionality, and consequently we reuse an existing *low-level* framework, LLVM. The LLVM infrastructure is an industrial-strength, open-source compiler toolchain modelling a portable representation of a register machine.

Introducing a middle-layer. The new platform serves as a middle-layer between an application's analysis and the low-level code generated for native execution. A different approach for supporting a variety of analyses would be a platform for *end-users*, putting a complete easy-to-use scripting language directly at their disposal. However, our primary goal is to enable the reuse of functionality across *applications*, and we argue that a single system can hardly address the needs of both users and applications simultaneously. For example, the Bro IDS provides a domain-specific language for traffic analyses yet it structures it along an event-based approach that would not fit well with the needs of, say, an application-layer proxy. In some sense, we see HILTI as a "low-level Bro" that provides much of its core functionality, yet does not tie an application to a specific analysis structure. HILTI is an ideal compilation *target* for an end-user language.

6 Summary

We introduce a middle-layer for building network applications, consisting of an abstract domain-specific execution environment (HILTI), and a compiler toolchain for turning programs written for that model into efficient, native code. We discuss the framework's design, and present an early prototype implementation that already supports many of its features. The key to achieving high performance is extensive, domain-specific code optimization, such as exploiting the enormous concurrency potential found in typical traffic analyses. We believe that the HILTI platform has the potential to become an established platform for a variety of applications, and will eventually enable our community to build a reusable library of robust yet efficient traffic analysis components.

References

[1] Parrot VM. `http://www.parrotcode.org/docs/`.

[2] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the Resource Consumption of Network Intrusion Detection Systems. In *Proc. Recent Advances in Intrusion Detection (RAID)*, 2008.

[3] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined Style. *SIGPLAN Notices*, 34(9):18–27, 1999.

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

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

[6] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. USENIX '93*.

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

[8] 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.