A Concurrency Model for Event-Based Network Intrusion Detection

Matthias Vallentin
vallentin@icsi.berkeley.edu

Seth Fowler
sfowler@eecs.berkeley.edu

1 Introduction

Network intrusion detection systems (NIDS) follow the communication in a network by inspecting the passing packet stream and raising alerts upon suspicious activity. The continuously rising traffic volume and the need to do more analysis at increasing complexities pose fierce performance challenges to these systems. Since NIDS operate in real-time, they cannot defer expensive computation to a later point in time. A typical deployment of a NIDS is illustrated in Figure 1.

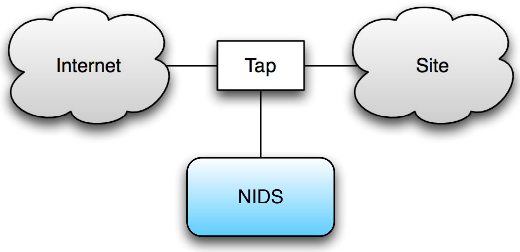


Figure 1: Typical deployment of a NIDS.

The required processing power to address these growing performance requirements cannot be provided by single-threaded, single-core architectures that are pervasive today. Performance improvements will instead come from many-core architectures which can only be harnessed by applications that feature a concurrent execution model.

Previous work identified that the key tasks in network security analysis bear a rich potential for parallelism [12]. In order to exploit this potential, it is crucial to understand the domain-specific workflow of a NIDS. Figure 2 visualizes the spectrum of parallelism that can be extracted in the network security analysis pipeline. In this figure, arrows represent the information flow from one stage to the next; the number of arrows leaving a particular stage corresponds to the data volume.

The first stage in the NIDS workflow dispatches packets to their corresponding flows. Since the arriving stream of packets is inherently sequential, one appealing solution is to implement packet demultiplexing in custom hardware [19].

The second stage operates on reassembled flows to analyze the application-layer protocol; it offers a high potential for parallelism. To see this potential, we have to appreciate that port numbers are not a reliable indicator for the protocol used in a connection. In order to reliably determine the actual application protocol, multiple instances of *all* potential analyzers can run in parallel [3] until the correct protocol is identified. As soon as the correct protocol is found, the one remaining analyzer parses the communication from there on as a series of policy-neutral events (e.g., HTTP request, TCP connection shutdown, DNS reply, etc.) that are forwarded to the detection logic.

These forwarded events are then processed at a *per flow* granularity; for example, locating downloaded executables in an HTTP stream. More complex forms of analysis – for example, detecting scanners by tracking the unique hosts a particular machine attempts to connect to – have a *per aggregation unit* granularity. In the scan detection example, the aggregation unit is the connection originator. The potential parallelism is bounded by the number of aggregation units and the extent to which they share state.

The final stage, *global* analyses such as stepping stone detection [21], offer the least degree of parallelism, as they operate across multiple aggregation units.

Parallel tasks in this pipeline keep their own state, yet share the same working data: multiple protocol analyzers execute concurrently but operate on the same connection. These properties map well to many-core architectures and allow for good cache locality by scheduling threads with shared working data to the same core.

Our contributions in this project are threefold. First, we design the necessary concurrency primitives that nat-

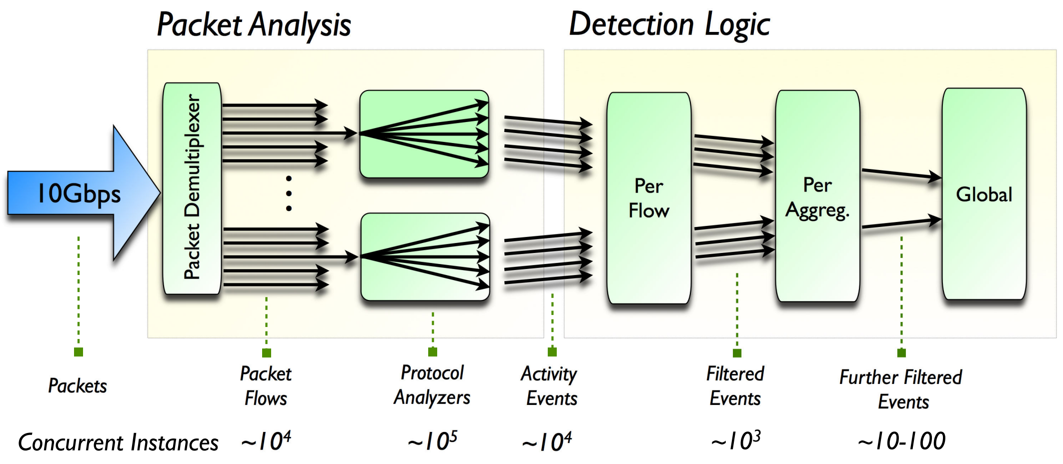


Figure 2: Spectrum of parallelism in the network security pipeline [15].

urally support the type of parallelism sketched above. Second, we implement these primitives in an abstract machine model. Third, we provide a quantitative evaluation of our implementation and elaborate on our experience with the LLVM toolchain.

The remainder of this paper is organized as follows. We briefly introduce the context of our project in §2. Thereafter, we describe our concurrency model in §3 and evaluate our implementation in §4. We point out limitations of our work in §5 and summarizing related work in §6 before we conclude in §7.

2 Background

This section provides the necessary background to understand the context of our project. After discussing the Bro NIDS in §2.1 we turn in §2.2 to HILTI, an abstract machine for network intrusion detection.

2.1 Bro

For the scope this project, we focus on the Bro NIDS [11] that features a rich-typed DSL to allow operators codifying their analyses in terms of its key semantic steps. The language has an asynchronous, event-based flavor that enables the expression of network policy at a high level of abstraction.

The implementation of the scripting language is quite complex due to its high-level nature, and the language is therefore interpreted. Together with the incurred overhead of interpretation, the limited opportunities for optimization render this a costly model compared to native code execution. Recent measurements witness indeed that Bro spends most of the time in the script interpreter.

2.2 HILTI

To overcome these critical limitations, a current research effort by the ICSI networking group is developing an abstract machine: the high-level intermediate language for traffic inspection, or *HILTI* . The overarching goal of this project is to create a generic platform geared towards the domain of network intrusion detection and supporting the field's common abstractions and idioms in its instruction set. One specific goal is bridging the gap between high-level abstractions to express a security policy and the optimized native code needed to achieve the required performance to operate on large-volume networks. [1]

HILTI is conceptually divided into two parts. On the one hand, the abstract machine serves as a compilation target. From this perspective, it is key to offer language constructs that identify high-level data parallelism, without exposing the underlying concurrency implementation.

On the other hand, the target for the abstract machine compiler is the open-source Low-Level Virtual Machine (LLVM) framework [4, 6], an industrial-strength compiler toolchain. By building on top of LLVM, many domain-independent standard optimizations can be leveraged to generate efficient code optimized for modern CPUs.

The implementation consists of a runtime and a compiler framework.[2] The runtime provides functionality that is either difficult to generate or requires unavailable instructions in LLVM. It is written in C but can be converted to bitcode using LLVM's GCC frontend.

[1]For the scope of this paper, we mostly restrict ourselves to the discussion of HILTI in conjunction with Bro, although HILTI is also beneficial in other network security related tasks.

[2]Since HILTI is still in a fledgling state, we spent a significant fraction of our time implementing runtime features and basic infrastructure.

The compiler transforms HILTI code into LLVM bitcode which can then be optimized and compiled in to a native executable. To reduce development time, the ICSI networking group chose to implement the compiler framework in Python. The available Python bindings [7] for LLVM facilitate constructing an in-memory representation of the generated LLVM code, making code generation much easier.

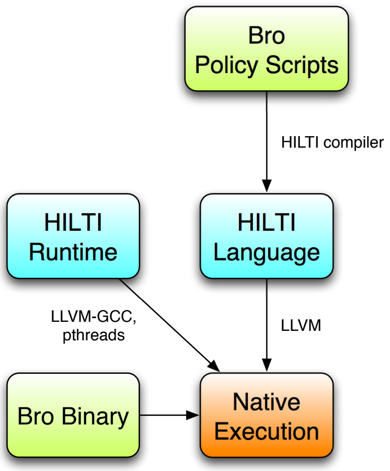


Figure 3: Components.

HILTI's relationship with Bro is visualized in Figure 3. A Bro-specific compiler translates policy scripts into the HILTI language; we hope to benefit from optimizations particularly at this point in particular, because the domain-specific semantics of the code are still preserved. In the next step, the HILTI compiler framework transforms the HILTI program into LLVM bitcode. The runtime library, written in C with pthreads, is also converted into LLVM bitcode using the GCC frontend for LLVM. In a final step, the Bro binary is linked statically, generating a native executable. If the host application is also compiled via the GCC frontend, LLVM can apply a variety of link-time optimizations across the entire executable.

3 Concurrency Model

Most of the parallelism available in a NIDS, as described above, conforms to the pipe-and-filter and event-based patterns. For aspects of the design which are pipelined, streams of data which are logically separate can flow through the pipeline in parallel, while individual pipeline stages for a particular stream must be processed sequentially. For the event-based components, there exists an analogous partial order over the events, such that related events are required to be processed in a particular order with respect to each other, while unrelated events are unconstrained. We therefore require a concurrency model that expresses partial orders over tasks in an efficient and scalable way. In §3.1, we give a concrete example of how such a concurrency model appears in one of the productivity-layer languages that HILTI supports. In §3.2, we describe the underlying HILTI concurrency model.

3.1 Concurrency in Bro

The multi-core version of Bro expresses concurrency implicitly. The programmer thinks of his or her script in terms of event handlers, each of which may access one or more shared variables. These shared variables are explicitly annotated with a *scope* which identifies the granularity at which they are shared; each event handler may only access shared variables from one scope. Scopes are identified in natural, domain-specific terms; for example, if a shared variable is in `connection` scope, all event handlers processing the same connection share the same copy of the variable. Bro includes predefined scopes for common cases, but the user may define new scopes if necessary. When event handlers share the same copy of a shared variable due to this mechanism, we say that the event handlers share the same *instance* of the scope, and that they operate in the same *execution context*. The execution context a particular event handler is running in may be different for each invocation, but the scope used to determine it is always the same.

The code fragment below demonstrates how scopes are used in a Bro script.

```
originator scans: set[addr];

event new_connection(c: connection)
{
    local responder = c$id$resp_h;
    add scans[responder];
}
```

In this code, `scans` is a shared variable with `originator` scope. It consists of a set of addresses; there will be a separate such set for each `originator` instance. The event `new_connection` is automatically placed in the originator scope since it accesses `scans`; the Bro runtime will place a specific invocation of `new_connection` in a particular execution context based upon the originator of the connection it will be operating upon. `new_connection` adds the responder of the connection to `scans`; the particular copy of `scans` is determined by its execution context.

The Bro concurrency model uses the execution context of an event handler invocation to schedule it. It guarantees that all events in the same execution context will execute serially, in temporal order. Events in different execution contexts, on the other hand, may execute in parallel, and no order of execution is guaranteed between them. Because scopes are determined by the shared variables an event handler accesses, and events in the same execution context execute in temporal order, no concurrent access to a shared variable is possible, and so no locks are required.

Bro offers some additional features related to scopes, such as the ability to override the automated classification process if it would place the event handler in the wrong execution context. Real-world experience of Bro users porting their policy scripts to multi-core Bro suggests that these features are unnecessary except in a few unusual situations. These porting efforts have also shown that scoped shared variables are sufficient to replace the vast majority of global variables in Bro scripts, allowing a great deal of parallelism with very little synchronization.

Most communication between execution contexts can be achieved using events: an event in one execution context can fire an event in another execution context, passing whatever parameters are necessary. In unusual cases, a Bro script may require a long-running event. Communication with this kind of event requires a different type of primitive. Channels, which are thread-safe, type-safe queues, serve this purpose. The details of channels in multi-core Bro are still under discussion, so we cannot provide any example syntax. Channels will be an integral part of other portions of the NIDS pipeline, such as protocol analysis, where they will provide the means for transmitting flows of network data between analysis components; the details are beyond the scope of this paper.

3.2 Concurrency in HILTI

HILTI's concurrency model is simple and explicit. Its unit of concurrency is the *virtual thread*, which consists of a queue of continuations that will be executed in serial order with respect to each other. Each virtual thread is identified by an integer in a HILTI program. HILTI functions may schedule function calls to execute on a particular virtual thread; these function calls will execute in the order that they are scheduled. Virtual threads may be run in parallel by the scheduler; in practice, each virtual thread is multiplexed on a *worker thread* shared by other virtual threads, with as many worker threads available as there are hardware threads in the system. The left-hand side of Figure 4 illustrates the mapping from execution context, through virtual and worker threads, down to hardware threads.

The scheduler currently in use is a simple stateless one that uses a Fowler-Noll-Vo hash function[10] to distribute the virtual threads as evenly as possible between worker threads. We have considered more complicated schemes that exploit state to assign the virtual threads actually in use to worker threads evenly, but any such scheme will require a potentially huge amount of state due to the large number of virtual threads that HILTI can support, and practical implementation constraints will prevent such a scheduler from operating in constant time, which is a serious flaw in the real-time environment of a NIDS.

Each virtual thread has *thread-local storage* associated with it. This storage may be used for variables which must be shared between the functions running on a virtual thread, such as the scoped shared variables discussed in §3.1; it is a distinct concept from thread-local storage at the worker thread level, since a single worker thread may be responsible for many virtual threads. The HILTI runtime automatically manages this storage; HILTI programs may simply refer to shared variables, and they will automatically receive the appropriate copy for the current virtual thread. One flaw to this system, however, is that thread-local storage cannot be collected by the HILTI garbage collector, since the garbage collector cannot tell when the program does not intend to schedule any more functions to a certain virtual thread that may depend on existing shared variables. For this reason, long-running programs will have to explicitly state that the thread-local storage on a certain virtual thread may be reclaimed. We are still discussing the detailed mechanisms by which thread-local storage will be implemented, and so it is not included in the current version of HILTI.

For long-running tasks that may need to communicate during execution, HILTI provides *channels*. These are thread-safe queues directly analogous to the ones discussed in §3.1. At the HILTI level, they are responsible for both the explicit uses of channels visible in the higher-level languages in the NIDS pipeline and low-level functions such as communication with hardware. Channels will hold the queue of continuations in each virtual thread in a future version of HILTI. They are also useful to implement features of the higher-level languages such as global variables which do not correspond to a concept in HILTI.

Because NIDS have to operate in real-time and must process packets at line-rate, it is important that each continuation executing on a worker thread finishes in a bounded amount of time. Worker threads implement cooperative multitasking: they execute their continuations sequentially, requiring very little overhead. This does not allow them to preempt a long-running continuation, which means that it is important that no continuation be allowed to block. This goal will be achieved by using channels for all I/O and all synchronous communication. Each attempt to read from a channel that would block in a conventional program will cause a new continuation to be scheduled on the same virtual thread

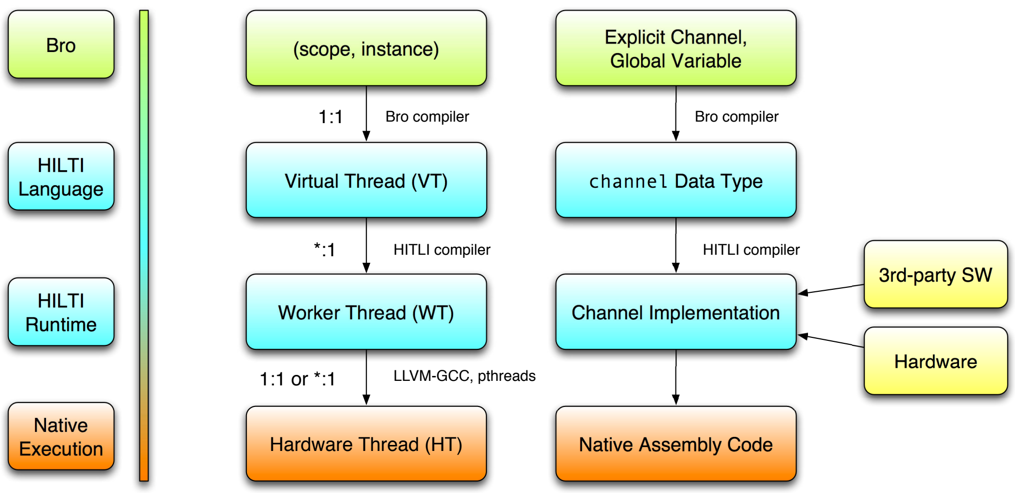


Figure 4: Concurrency stack.

in a special queue aside from the main one. Any other continuations which are scheduled to the same virtual thread will be moved to this special queue as they are encountered; this preserves HILTI's concurrency semantics. When the channel is able to fulfill the request, it notifies the worker thread, which returns the continuations in the special queue to the main queue. Special queues are created as needed whenever virtual threads would ordinarily block. We are currently working on implementing these features; although channels are currently working, their interaction with the scheduler was not ready in time for this paper.

4 Evaluation

This section presents our preliminary evaluation. After explaining our methodology in §4.1, we discuss our findings in §4.2.

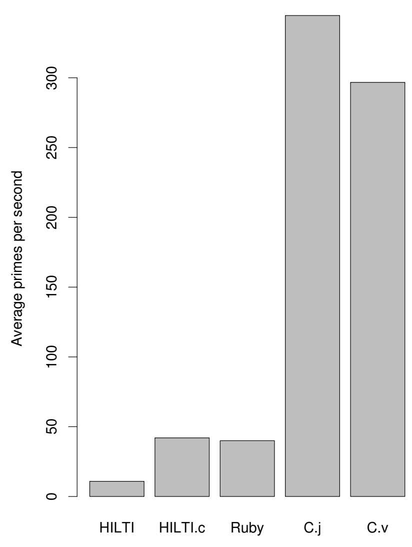
4.1 Methodology

Since HILTI is designed for problems which are naturally pipelined or event-based, we selected the Sieve of Erastothenes[20] as a suitable algorithm to benchmark. In the Sieve of Erastothenes algorithm, prime numbers are generated using a pipeline approach. At each stage of the pipeline, a list of integers is received; the first number in the list is the current prime. Every number in the list which is divisible by the current prime is removed from the list, and the remaining numbers are passed on to the next stage in the pipeline. The input to the first stage is a list of integers starting at 2 and ending at the maximum number of interest. After the algorithm has terminated, the Sieve of Erastothenes will have found every prime number in this range.

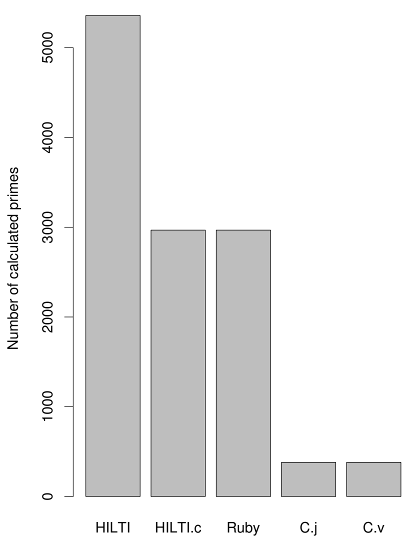
We implemented the algorithm in three languages: HILTI, C, and Ruby 1.9. In HILTI, the algorithm is most naturally implemented by having a virtual thread for each prime (that is, each pipeline stage) and scheduling a function on that virtual thread for each number that pipeline stage must process; knowing the prime number corresponding to the virtual thread it's running on, the function simply passes the number it receives to the next stage in the pipeline (and hence the next virtual thread) if the number is not evenly divisible by the current prime. To process a large range of numbers, a huge number of virtual threads must be used and a huge number of functions must be scheduled, which makes this algorithm an excellent test of HILTI's concurrency infrastructure.

There is one problem with this implementation: thread-local storage must be used so that each virtual thread knows what prime it corresponds to. Since we have not yet implemented this feature, we simulated it in a very simple manner using C functions that set and get key-value pairs stored on a linked list; this is dramatically slower than our eventual thread-local storage solution will be. In addition, garbage collection is not yet enabled for HILTI, which means that the memory allocated for each continuation cannot be reclaimed.

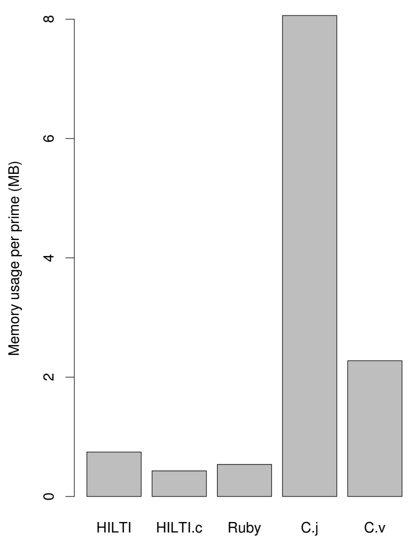
In C and Ruby, which do not have a notion of virtual threads, we implemented the algorithm using the standard threading facilities for each language (pthreads for C, the Thread class for Ruby) and made use of one thread per pipeline stage. Passing numbers between stages is accomplished using thread-safe queues. Unlike the HILTI version, the C and Ruby versions had to manually manage their threading and communication resources. We created two versions of the C program; C.j uses `pthread_join` to ensure that all threads have ter-



(a) Average primes per second.



(b) Number of calculated primes until termination.



(c) Memory usage per prime (MB).

Figure 5: Preliminary results.

minated, while C.v uses a volatile variable to check for termination. C.v was created because the original benchmark, C.j, runs out of thread resources so quickly.

We performed the benchmarking on an AMD Athlon X2 BE-2400, a 2.3 GHz dual-core machine with 2GB of RAM. Each program started with a list of numbers from 2 to 50,000 and was allowed to run until it generated all of the prime numbers in that range. Since the HILTI implementation was the only one able to finish without crashing, we increased the upper end of its range slightly, to 52,500, which was sufficient cause it to crash as well. This enabled us to see the point of failure for all of the implementations. It would have been better to use exactly the same range for all of the implementations, but we did not notice this problem until it was too late to correct; we don't expect the results to be substantially effected.

To calculate average running time, we executed each program for ten consecutive runs. Memory usage was calculated using the RSS column of `ps`'s output at the instant each program crashed; the program was frozen at that point using a `gdb` breakpoint. Memory usage monotonically increased during program execution and was essentially constant between runs for each program. The number of primes calculated is the maximum each program achieved before crashing; it was also essentially constant between runs, despite the parallel nature of each program, because the crashes were usually due to running out of resources such as threads which are almost perfectly correlated with the number of primes successfully calculated in this algorithm.

4.2 Results

Figure 5b displays the total numbers of prime numbers that each implementation was able to calculate before crashing. The HILTI implementation was able to calculate almost 81% more primes than its closest competitor, the Ruby implementation. It finally terminated due to running out of heap space after allocating almost 4GB of memory. The remaining benchmarks all terminated due to exhausting their supply of threading resources: in the Ruby case, `Thread.new` threw an exception, while in the case of the C implementations `pthread_create` returned `EAGAIN`. As mentioned in §4.1, we created C.v because C.j ran out of threading resources so quickly; however, C.v crashed after performing about the same amount of work. This indicates that the C version creates new threads much faster than it can process the work for each thread; a different design could have avoided this problem, but it would no longer look much like the simple HILTI design.

In Figure 5c, we see the amount of memory that each implementation required per calculated prime. The C implementations appear to require dramatically more memory, but this is misleading. While there may be some inefficiency in the thread-safe queue implementation we used in the C versions, the vast majority of the discrepancy is caused by overhead. The Ruby and HILTI implementations ran long enough for the overhead of each program to be amortized over a very large number of primes, while the C implementations died almost immediately. HILTI required slightly more overhead per prime than Ruby; this is attributable to a combination of HILTI's garbage collection not yet being enabled and the inherent overhead of creating a HILTI continuation for each number in each pipeline stage. The lack of garbage collection is the larger problem; by the end of the HILTI program's run, the vast majority of memory is being occupied by dead continuations.

The number of primes each implementation calculated per second is shown in Figure 5a. The C implementations, unsurprisingly, are the fastest by far. The Ruby version is an order of magnitude slower, and the HILTI version's speed is a quarter of that. This result initially surprised us, as it did not correspond to our intuition. However, we noticed that the HILTI version has allocated almost 4GB when it crashes, which is more memory than is physically present in the machine. We suspected that the memory leak caused by the lack of garbage collection was causing the HILTI version to use swap space, which was slowing it down. To test this theory, we implemented a revised HILTI version, HILTI.c, intended for comparison against the Ruby version. HILTI.c stops running after it has calculated the same number of primes that the Ruby version calculated, enabling us to compare the two after they have performed the same amount of work.

Figure 5a shows that HILTI.c is slightly faster than the Ruby version; it computes 42.00 primes per second as compared to 39.98 with Ruby. Figure 5c shows that HILTI.c also uses less memory per prime computed than the Ruby implementation, which is quite surprising since it leaks memory so badly. Considering that the HILTI version schedules a new function on a virtual thread and executes it for each number in each pipeline stage, a tremendous amount of overhead, it is quite impressive that it was able to outperform the Ruby version in terms of both speed and memory usage. This is a very encouraging result for us, since even with HILTI extremely unoptimized and missing crucial components, it is able to perform at the level of a popular language which has existed for many years. We are confident that with further work we will be able to improve dramatically improve HILTI's efficiency.

5 Limitations

We acknowledge that our work has some limitations. In order to support freezing the current execution state of functions to resume them at a later point of time, HILTI function calls are heap-allocated continuations [1] implemented in continuation passing style (CPS). Unfortunately, as mentioned in §4.1, the garbage collector is not yet wired into the compiler framework. Hence memory allocated on the heap is not freed, which explains the high footprint experienced in our evaluation.

Moreover, Python bindings for LLVM currently do not offer an option to enable tail-call optimizations (TCO). However, HILTI's execution model fundamentally relies on the LLVM optimizer to eliminate tail calls. Although a patch we recently located fortunately provides the missing functionality, for the majority of the project's duration we had to manually edit the extremely verbose machine-generated LLVM intermediate representation files if we wanted TCO to be applied successfully. We are still unable to enable TCO between different compilation units, since enabling LLVM link-time optimizations of any kind causes linking to fail. Our investigation of this issue is ongoing.

6 Related Work

Parallelizing network security analysis is a multi-faceted undertaking, since the various angles allow for distinctive approaches. The popular Snort NIDS [13] is based on a *signature matching* engine, where regular expressions match byte-wise on packet streams. Offloading pattern matching to FPGA-based or custom hardware is a common approach to gain further performance improvements [14, 5, 8, 2]. However, Moore's Law does not hold for these customized solutions. Not only do network processors evolve at a much slower pace, but newer incarnations often break code compatibility to older versions, forcing developers to reimplement a significant fraction of their code. In addition, it is difficult express composable high-level analyses with the available low-level primitives which are geared towards a *stateless* execution model.

There also exist approaches to gain speed-up by executing multiple Snort instances in parallel [18, 9] or by offloading parts of the analysis to the GPU [17]. We believe that HILTI poses an attractive alternative method to compile Snort rules into platform-specific code.

Current efforts to parallelize the single-threaded version of Bro are inspired by our previous work in [16], where we built a *NIDS cluster* on commodity hardware to load-balance connections across a set of communicating Bro instances. The stateless packet dispatcher forwards packets at Gbps rates and will replace the hitherto single-thread Bro versions at Lawrence Berkeley National Laboratory in the near future. Further, a Bro cluster of approximately 30 machines will soon monitor the traffic of the entire UC Berkeley campus.

7 Conclusion

The steadily growing traffic volume network intrusion detection systems need to monitor imposes high performance requirements, as these systems need to operate in real-time and have to perform both more and more sophisticated tasks. With the collapse of Moore's Law for single-core architectures, NIDS have to gain their performance from tomorrow's many-core CPUs.

In this project, we designed the necessary toolkit to harness the available parallelism in the network security pipeline [11]. The primitives we provide form the parallel building blocks of the HILTI abstract machine, a versatile execution model and compilation target for network traffic applications.

While HILTI contains to be in an early stage of development, even at this stage we were able to best Ruby in terms of speed and memory usage for an identical workload. We are encouraged by these results and feel confident that further development of HILTI will make implicitly parallel productivity languages like multicore Bro a reality in the field of network intrusion detection.

We expect further performance improvements by switching to lock-free data structures in the scheduler and channel implementation, and by enabling garbage collection. Most notably, our project lays the foundation stone for future work, of which a major part is the implementation of a HILTI compiler for the Bro scripting language. The provided primitives enable a natural translation of the scoped execution model to HILTI.

References

[1] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, New York, NY, USA, 1992.

[2] Young H. Cho and William H. Mangione-Smith. Deep Network Packet Filter Design for Reconfigurable Devices. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–26, 2008.

[3] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. Dynamic Application-Layer Protocol Analysis for Network Intrusion Detection. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[4] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proc. International Symposium on Code Generation and Optimization*, 2004.

[5] Janghaeng Lee, Sung Ho Hwang, Neungsoo Park, Seong-Won Lee, Sunglk Jun, and Young Soo Kim. A high performance NIDS using FPGA-based regular expression matching. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1187–1191, New York, NY, USA, 2007. ACM.

[6] The LLVM Compiler Infrastructure. `http://llvm.org`.

[7] llvm-py: Python Bindings for LLVM. `http://mdevan.nfshost.com/llvm-py`.

[8] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for Accelerating SNORT IDS. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 127–136, New York, NY, USA, 2007. ACM.

[9] NinjaBox-Z and Applied Watch. `http://www.endace.com/assets/files/ninjaBoxZ_applied_watch.pdf`.

[10] Landon Curt Noll. Fowler / Noll / Vo (FNV) Hash. `http://www.isthe.com/chongo/tech/comp/fnv/index.html`.

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

[12] Vern Paxson, Krste Asanovic, Sarang Dharmapurikar, John Lockwood, Ruoming Pang, Robin Sommer, and Nicholas Weaver. Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In *Proc. USENIX Workshop on Hot Topics in Security*, 2006.

[13] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *In Proceedings of the Systems Administration Conference*, 1999.

[14] Reetinder Sidhu and Viktor K. Prasanna. Fast regular expression matching using fpgas. In *FCCM '01: Proceedings of the the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society.

[15] Robin Sommer. Exploiting Multi-Core Processors For Parallelizing Network Intrusion Prevention. TRUST Seminar Series, UC Berkeley, May 2009.

[16] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The NIDS Cluster: Scalably Stateful Network Intrusion Detection on Commodity Hardware. In *RAID*

'07: Recent Advances in Intrusion Detection, 10th International Symposium, Lecture Notes in Computer Science, pages 107–126. Springer, September 2007.

[17] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 116–134, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] Javier Verdú, Mario Nemirovsky, and Mateo Valero. MultiLayer processing – An Execution Model for Parallel Stateful Packet Processing. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 79–88, New York, NY, USA, 2008. ACM.

[19] Nicholas Weaver, Vern Paxson, and Jose M. Gonzalez. The Shunt: an FPGA-based Accelerator for Network Intrusion Prevention. In *FPGA '07: Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, pages 199–206, 2007.

[20] Eric W. Weisstein. Sieve of Eratosthenes. `http://mathworld.wolfram.com/SieveofEratosthenes.html`. Wolfram Mathworld.

[21] Yin Zhang and Vern Paxson. Detecting Stepping Stones. In *SSYM'00: Proceedings of the 9th conference on USENIX Security Symposium*, pages 13–13, Berkeley, CA, USA, 2000. USENIX Association.