Beyond Pattern Matching: A Concurrency Model for Stateful Deep Packet Inspection

Lorenzo De Carli
Dept. of Computer Sciences
Univ. of Wisconsin, Madison
lorenzo@cs.wisc.edu

Robin Sommer
ICSI / LBNL
Berkeley, CA, USA
robin@icir.org

Somesh Jha
Dept. of Computer Sciences
Univ. of Wisconsin, Madison
jha@cs.wisc.edu

ABSTRACT

The ever-increasing sophistication in network attacks, combined with larger and larger volumes of traffic, presents a dual challenge to network intrusion detection systems (IDSs). On one hand, to take advantage of modern multi-core processing platforms IDSs need to support *scalability*, by distributing traffic analysis across a large number of processing units. On the other hand, such scalability must not come at the cost of decreased effectiveness in attack detection. In this paper, we present a novel domain-specific concurrency model that addresses this challenge by introducing the notion of *detection scope*: a unit for partitioning network traffic such that the traffic contained in each resulting "slice" is independent for detection purposes. The notion of scope enables IDSs to automatically distribute traffic processing, while ensuring that information necessary to detect intrusions remains available to detector instances.We show that for a large class of detection algorithms, scope can be automatically inferred via program analysis; and we present scheduling algorithms that ensure safe, scope-aware processing of network events. We evaluate our technique on a set of IDS analyses, showing that our approach can indeed exploit the concurrency inherent in network traffic to provide significant throughput improvements.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network monitoring*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

Keywords

NIDS, Flexible intrusion detection, Scalable traffic analysis

1. INTRODUCTION

Effective network intrusion detection is becoming increasingly difficult. With the proliferation of connected devices and web-based services, network bandwidths keep soaring, putting stringent performance requirements on detectors that must sift in real-time through large data volumes. Moreover, the nature of network intrusion itself is evolving, driven by an emerging underground economy and the rise of resourceful, nation-level adversaries ("Advanced Persistent Threats" [18]). As attack strategies are shifting from conceptually simple byte-level exploits to sophisticated, tailored attacks operating deep inside the application layer, intrusion detection systems (IDSs) need to adapt to remain effective and relevant. In order to scale to larger volumes of traffic, they must support *concurrency* to take advantage of modern multi-core architectures. Yet, at the same time, the increased complexity of attack strategies requires *flexibility*, as no one-size-fits-all approach to detection will prove effective against the modern arsenal of attack tools.

Unfortunately, there exists a fundamental tension between these two objectives. Simple, static detection strategies bring predictable data flows and inter-thread communication, which allow to "hard-code" efficient parallelism into the IDS design. An example is signature matching: As signatures are commonly expressed on a per-flow basis, an IDS performing this operation can simply process each connection independently. Yet, signature-based detection remains limited in expressiveness, and can often be thwarted with minor changes in the attack strategy—consider the fragility of the early signatures for the Heartbleed bug [8], or the use of binary obfuscation to make malware undetectable [7]. Avoiding such limitations requires more complex strategies, including stateful protocol analysis and correlation of events across multiple flows. That complexity, however, turns parallelization into a much harder problem.

The current state of mainstream IDSs reflects this tension. Suricata [11] and Snort [10] support multi-threaded processing (the latter through a variety of different proposals, e.g., [43, 45, 48]), but they remain limited to classic per-flow signature matching. Other related efforts in the literature [28, 44] rely on specialized hardware and/or similarly hardcoded detection algorithms. To our knowledge, Bro [6] represents the only IDS that offers complete flexibility by design; it expresses detectors in a Turing-complete scripting language. However, Bro remains single-threaded to this day.

Our work presents a step towards making intrusion detection both parallel *and* flexible. We propose a *general concurrency model for network traffic analysis* that can guide IDS architectures towards parallel performance, independent from the underlying detection strategy. To detach our model from the specifics of a detector, we focus on generic data-level parallelism, as opposed to process-level parallelism (e.g., pipelining) as that remains heavily implementation-dependent. We observe that network traffic is in fact inherently parallel: typical 10 GE upstream links routinely carry 100,000s of active flows that reflect the communication of mostly unrelated endpoints. In other words, analyzing network traffic constitutes an *almost* "embarrassingly parallel" task [37]. However, while flows generally proceed independently, most of them also share a close semantic relationship with *some* of the other ones—which an IDS must account for. Consider the activity that

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660361.

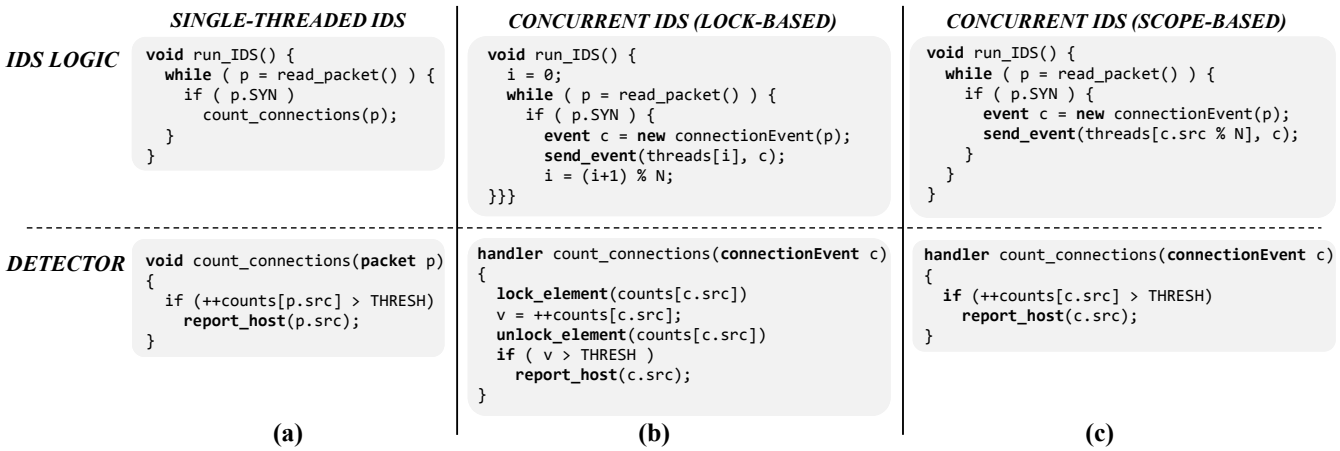


Figure 1: Simple portscan detector

is part of the same user's browsing session, or traffic generated by an attacker slowly scanning a target network for reconnaissance. The latter may resemble a series of innocent requests, without any recognizable fingerprint, and the attack would manifest itself only to a detector that maintains connection statistics for each possible source over a long period of time. An IDS, hence, needs to sift through a large number of flows, mostly unrelated, while correlating the minuscule fraction that reveals the malicious activity.

For devising a general concurrency model, we start from the observation that packet processing is by nature event-driven, with events representing semantic units of protocol activity (e.g., the establishment of a new session, or, at higher-level, an HTTP request). Events typically trigger a simple computation that often accesses, and potentially modifies, persistent data structures tracking the analysis' current state. We find this abstraction sufficiently generic to encompass the semantics of all popular IDS applications. We then formalize the concepts of processing *scope* and *state*: scope represents a unit for partitioning network traffic such that processing of each slice remains independent from the others, and hence may proceed in parallel with them; and state refers to the aggregate information that associates with computations operating at a scope's granularity. Consider a simple scan detector, counting connection attempts by source: it operates with a scope of "source IP address", and its state comprises the table that maps addresses to counter values. As each counter depends solely on the activity of the associated source address, we can "slice" both computation and state at the scope-level (i.e., IP addresses) to parallelize the detector without further inter-thread communication.

From the perspective of this model, signature-based IDSs tend to have a single program-wide scope (e.g., flows in Suricata) and hence enable deploying a specific hard-coded slicing strategy (e.g., per-flow load-balancing). Once we allow for more complex analysis paradigms, on the other hand, it becomes impossible to identify just a single scope and thus optimize the implementation accordingly. For example, in Bro every analysis script may structure its processing differently, and hence require a separate scope. Our work identifies *all* relevant scopes statically at compile-time by using a novel application of *program slicing* (§4). We then use the information to drive a dynamic thread scheduler at run-time.

To demonstrate our approach we implement it inside a generic IDS middle-layer platform that provides a set of domain-specific programming constructs for expressing arbitrary network analysis tasks. We find this approach effective in achieving scalability (§6).

We structure the remainder of the paper as follows: §2 develops our concurrency model and the notion of scope; §3 discusses how to generalize this notion to complex detection strategies. §4 discusses how to infer scope from IDS programs via static analysis, and §5 formally defines a scope-aware event scheduler. §6 presents experimental results, §7 discusses limitations of our approach, §8 presents related work, and §9 concludes the paper.

2. IDS CONCURRENCY MODEL

Modern hardware architectures offer plenty of parallelism to address scalability concerns [5, 9]. Unfortunately, current mainstream IDSs either do not take advantage of these parallel platforms, or in doing so restrict their capability to simple, hard-coded detection strategies, thus limiting flexibility.

Part of the problem is the lack of a clear definition of which detection strategies a parallel IDS should support, and what should be its concurrency model. In our work, we approach this issue by (i) inferring a domain-specific but flexible model of how IDSs process traffic, and (ii) leveraging this model to define a practical IDS concurrency model.

2.1 Reference IDS

Before discussing a concurrency model, it is important to define the structure and capabilities of our IDS. For the purpose of our work we use an abstract IDS model based on Bro, whose flexible structure fits our goal of constraining analyses as little as possible.

The first idea we mutuate from Bro is a clear architectural separation between fixed, low-level packet processing tasks ("mechanism")—such as checksum verification, stream reconstruction, protocol parsing etc.—and the detection task proper ("policy"). Specifically, the lower layer generates a stream of pre-digested events for the higher layer to analyze.[1] In order to achieve a fully parallel IDS, both layers—low-level traffic processing and high-level analysis—must be parallelized. There is a significant body of work showing that low-level traffic processing can be efficiently parallelized at connection granularity. Relevant approaches include the NIDS cluster [40], novel IDS proposals such as Kargus [40] and Midea [44], and various efforts to parallelize Snort [43, 45, 48]. Taken together, these results enable us to conclude that low-level traffic processing—as well as intra-connection detection—can be efficiently parallelized at connection granularity, scales well in practice, and

[1]Events can represent occurrences at all layers of the protocol stack, thus not limiting detectors to a specific level of abstraction.

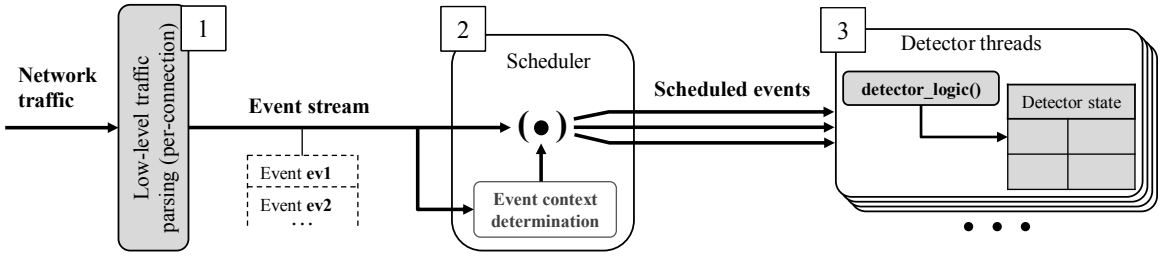


Figure 2: High-level IDS architecture

does not represent the main challenge for a distributed IDS. Therefore we focus on parallelization of high-level, inter-connection events.

In this context, our IDS allows users to define detection strategies as event handlers expressed in a Turing-complete domain-specific language. This approach encompasses the semantics of virtually all popular IDS platforms, enabling our concurrency model to retain generality. For the purpose of this paper, we express analyses using a C-like syntax with a few IDS-specific primitives and data types (see Figures 1, 8).

2.2 Event-based Concurrency

A natural approach to parallelization is to distribute events to an array of IDS threads. The difficulty here is that high-level intrusion attempts (and related behaviors) are typically fingerprinted by multiple *correlated* events. For example, consider the simple portscan detector in Figure 1(a). The upper half of the figure describes the event-generating logic ("IDS logic"), while the lower half describes the detection algorithm. Albeit admittedly contrived, this program adheres to our IDS model and works by correlating multiple consecutive events (connection attempts from a given host).

A strawman parallel version may look like the program in Figure 1(b). This implementation works by generating one event for each new connection; events are fed to a pool of N identical threads in round-robin for processing. This example illustrates a fundamental issue: most detectors—even very simple ones—maintain a certain amount of *state* that is progressively updated as events are processed. The main problem of our strawman implementation is that events assigned to different threads are not independent—different threads may end up processing events related to the same source. Therefore, access to the state of the detector must be mediated by locks ("lock_element()" in the example) to avoid data races. Similar to conventional approaches for general-purpose programs, the parallel behavior is hardcoded in the script, and data races are avoided by using costly synchronization primitives. Moreover, the program in Figure 1(b) will in general route multiple events from the same source to different threads. This causes each thread to perform a sequence of accesses with little or no memory locality; if the amount of state kept by the detector is significant, continuously retrieving and updating unrelated pieces of state can severely hamper performance. Finally, this approach does not preserve ordering of events. While this is irrelevant for our example, many real-world IDS analyses (e.g., ones that correlate a sequence of malicious actions) are in fact sensitive to re-ordering.

A key insight about IDS analyses [30, 37] is that, even when no particular constraints are imposed, they tend to naturally structure themselves around independent units of processing—such as flows, hosts, subnets etc.—and to access little or no state outside their unit of processing. For example, code that examines the content of a particular flow rarely requires access to information about *other* flows; and our example scan detector has no need for correlating counters between sources. In other words, *partitioning events by unit of processing also partitions the detector state in independent subsets*. In the rest of this paper we refer to a unit of processing and its related state as the *scope* of the detector. Also, we refer to a concrete instantiation of a scope as a *context* (e.g., if the scope is "connection", a context is a concrete instantiation of the 5-tuple).

Our concurrency model requires a scope to be associated with each analysis. Said scope defines a contract between an analysis and the underlying IDS runtime, where the analysis "promises" to only access state within its scope. In exchange the runtime provides the following guarantee: *all network events within the same context are processed sequentially by the same thread, in the order they are received*. For our simple scan detector, the scope is the source address (*c.src*). Figure 1(c) depicts its implementation within this paradigm: each connection is statically mapped (by simple hashing, $c.src\%N$) to one of the available threads, guaranteeing that (i) no two threads access the same state at the same time, and (ii) all events from the same source are processed sequentially. It should be noted that scope can be non-trivial to define, especially for analyses aggregating multiple connections at the application layer. §3 presents one such analysis (a worm detector), and discusses an approach to generalize the notion of scope to those cases.

2.3 A Parallel IDS Architecture

We now outline a concrete IDS architecture based on the concurrency model discussed above. We have implemented and evaluated this architecture; results are discussed in §6.

Our proposed architecture, depicted in Figure 2, assumes a preprocessing step (1) to efficiently parse raw packets and generate events (in the case of our example portscan detector, new connection notifications). A scheduler (2) then determines the appropriate context for each event (the address of the connection originator), and maps all related processing to the corresponding thread. The resulting stream of scheduled events is analyzed using multiple *analysis threads* (3), each in charge of a set of contexts. Each thread maintains and updates its own private, local detection state.

The scalability of this model depends crucially on finding sufficient diversity in the analyzed traffic (in terms of number of contexts) to distribute load and state evenly across threads. Previous work has shown that partitioning traffic at flow level [40] and similar units [37] balances well and provides good thread-scalability. Our evaluation, presented in §6, supports these conclusions.

Our model relies on (i) the availability of a well-defined scope for each detection strategy, and (ii) the correctness of the event scheduler. In §3 we discuss how scope can be generalized using the concept of *scheduling functions*, and in §4 we propose an approach to automatically infer scope via program analysis. §5 then gives a scheduling algorithm suited for our architecture.

3. GENERALIZING DETECTOR SCOPE

For simple analyses, the processing context of an event handler is directly characterized by its input data. For example, in the

portscan detector of Figure 1 the context is given by the address of the connection originator. Similarly, for a detector performing per-flow signature matching each event's context is determined by its connection 5-tuple. Therefore, it is tempting to specify scope as a subset of input parameters (such as *c.src* for the detector of Figure 1(c)).

This assumption however does not hold for more complex analyses, that may correlate multiple flows and different classes of network events. In this section we demonstrate the issue using a simple worm detector, and we show how to achieve a more general definition of scope via the concept of *scheduling functions*.

3.1 *Multistep*: a Trojan Detector

Malicious network activity by an infected host tends to consist of various operations that appear normal if considered individually but become significant once considered together. Our sample analysis implements a simple multi-step trojan detector (*multistep* in the following), inspired by publicly available Bro didactic material [1]. Albeit referring to a fictional malware, it is inspired by threats seen in practice, making it a realistic case study.

The target of the detector is a backdoor application that is associated with the following sequence of operations: (i) the infected host receives an SSH connection on port 2222; (ii) the host initiates three distinct downloads: an HTML file from a web server, and a ZIP and an EXE file from a FTP server; (iii) the host generates IRC activity. Note that order is relevant; the same events in a different order do not constitute a fingerprint.

We assume the availability of an underlying IDS layer that can distill raw packet traffic into high-level events, as described in §2.1. These events are fed to the detection logic, which consists of three event handlers:

- *ProtocolConfirmation*: Triggered by the IDS when an application-level protocol is being used within a connection. Used to detect both the initial inbound SSH connection, and the final outbound IRC connection.
- *HttpRequest*: Triggered when a host generates an HTTP request. Used to detect the HTTP download.
- *FTPRequest*: Used to detect both FTP downloads.

To maintain state the detector uses a persistent table, consisting of an associative container indexed by IP addresses of potentially infected hosts. The value associated with each IP is the current detection state i.e., how many actions, from the sequence that fingerprints the trojan, the host has already performed. An entry is created in the table for each host that receives an SSH connection on port 2222, and updated every time the same host performs one of the activities described earlier. If a host completes all the actions in the described order, the detector raises an alert.

3.2 Parallelizing *Multistep*

To determine the appropriate scope for *multistep*, we begin by considering how data flows within an individual event handler, summarized in Figure 3(a). Events from the input stream (1) are fed to event handlers (2) as they arrive. Each handler derives a key from input data (3), per the labels on the edges; and then uses that key as an index to retrieve relevant detection state (4). First, just by considering the inputs to each handler, it is evident that a per-flow approach to parallelism is infeasible, since the various event handlers operate on different connections. If we look for another, more general scope to partition the traffic (connection originator? connection responder?), a problem quickly becomes apparent: Each

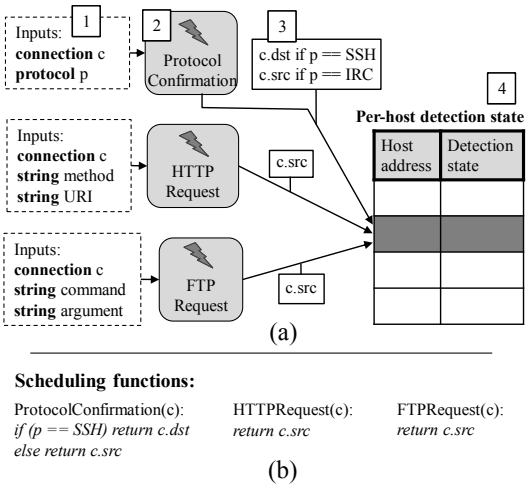


Figure 3: Dataflows (a) and scheduling functions (b) for multistep

handler *independently derives the index used to retrieve the detection state*. As can be seen, there is no unique way to define a scope that applies to all the components of the application, since the information of interest can be either the originator of a connection, or the responder. It is not even possible to assign a well-defined scope to individual handlers: in the case of the *ProtocolConfirmation* handler, the index can be either the connection originator or responder depending on which protocol is being detected.

These considerations suggest that attempting to define scope from a network perspective, i.e., statically and in terms of protocol-related concepts (flow, connection originator, etc.), is not suitable for cross-layer, complex analyses. Instead, we propose considering the issue from an "analysis-centric" point of view: *all the possible inputs that cause the same detection state to be accessed belong to the same context*. In other words, we make the definition of scope dependent on the computation performed by the program itself.

3.3 A Flexible Approach to Scheduling

We observe that most analyses are structured around a set of tables, and their persistent state is fully defined by the values of the indices used to access said tables. Consider the *ProtocolConfirmation* handler in Figure 3(a). The handler is executed each time a new connection is observed, and receives an identifier p for the protocol being used. If $p == IRC$, the handler accesses the table based on the source of the connection (*c.src*). If $p == SSH$, it does the same using the destination of the connection (*c.dst*). The key point is that *once the table is accessed, the scope gets fully disambiguated.*

This suggests a way to conceptually partition a set of events into contexts: two events are within the same context if they cause the detector to access the same index (indices) in its table(s). For example, an IRC connection *from* address 192.168.1.12 and a SSH connection *to* the same address will both cause *multistep* to update the same table entry. At the same time, events generated by a another infected host/IP will affect a different entry.

This rule can be directly used for scheduling, by mapping all events resulting in the same table access(es) to the same context, and therefore to the same thread. But there is a caveat: the value of table indices can only be determined when the event handler executes, i.e., after the scheduling decision has been done. However, a large number of event-driven analyses, regardless of their complexity, statelessly compute table indices from the values of their input parameters (event data).

Our approach then consists in annotating each event handler with its index computation, which we call the *scheduling function* $\mathcal{A}$. Figure 3(b) outlines the simplest possible scheduling functions for the handlers in the *multistep* example. The role of $\mathcal{A}$ is to guide scheduling by deriving the scope from input values for each new event, before processing it. Once scheduling functions are available, parallelization can proceed by executing the appropriate scheduling function on each input, and using the result to map events within the same context (i.e., accessing the same data) to the same thread. As Figure 3(b) illustrates, expressing scope in terms of scheduling functions does not introduce additional overhead: a minimal scheduling function expresses precisely the operations that the IDS logic must perform to derive the appropriate context for an event.

In §4 we give algorithms to automatically construct an efficient scheduling function $\mathcal{A}$ for a given program P via static program analysis. An advantage of this technique is that the parallelization strategy is derived offline, i.e., scheduling functions can be fully constructed before running the program.

4. INFERRING SCOPE

The approach outlined in §3 requires, for each analysis, a *scheduling function* $\mathcal{A}$. The simplest approach for generating $\mathcal{A}$ is to require the developer to annotate each program with an appropriate scheduling function. This is however impractical, as it further complicates the user's already difficult task of implementing effective traffic analyses. Developing an analysis and the corresponding scheduling function is cumbersome, and the duplication of code with similar purpose makes programming errors more likely. If a program and its scheduling function become inconsistent, the analysis risks incurring false negatives. Moreover, reasoning about scheduling functions requires the user to focus on a technical aspect of the system—parallelization—unrelated to the main goal of intrusion detection. Instead, our goal is to provide an IDS system with transparent scalability, leaving the user free to concentrate on developing effective analyses.

We therefore consider the problem of automatically generating the scheduling function $\mathcal{A}$ for a given IDS program P. If $Ind(i)$ is the set of indices accessed by P, $\mathcal{A}$ is defined so that $Ind(i) \subseteq \mathcal{A}(i)$ for all $i \in \mathcal{I}$ (where $\mathcal{I}$ is the set of all possible program inputs). We begin by observing that the most obvious definition of $\mathcal{A}$ is P itself. Making $\mathcal{A}$ equal to P results in a scheduling function that is fully precise, since for every input i it always returns exactly the set of indices $Ind(i)$ that P will access. However, such $\mathcal{A}$ is also terribly inefficient, as it causes P to run twice on each input: first to perform scheduling and then to process the event. The problem then becomes to construct $\mathcal{A}$ an as an *over-approximation* of P, such that $Ind(i) \subseteq \mathcal{A}(i)$ and $\mathcal{A}$ executes faster than P.

To construct such an approximation, we observe that for many IDS heuristics only a small part of the program is dedicated to computing the indices in $Ind(i)$, while the rest implements the detection logic. Therefore, a compact (with respect to the size of P) scheduling function $\mathcal{A}$ can be obtained by pruning all the statements, in P, that are irrelevant for the computation of $Ind(i)$. In the rest of this section, we describe static analysis algorithms that constructs the scheduling function $\mathcal{A}$ by pruning P. As both the algorithms are based on *program slicing*, we provide a brief primer.

4.1 Program Slicing Primer

Program slicing [27, 35, 47] is a program analysis technique that provides two primitives: (i) determine which statements in a program influence the value of a variable at a given point (*backward slicing*), and (ii) determine which statements are influenced by the value of a variable at a given point (*forward slicing*). It does so by leveraging the *program dependency graph* (PDG), a graph representation of a program where nodes are statements and edges represent data and control dependencies between statements. Thus for example backward slices can be constructed by computing backward reachability from the statements of interest. Figure 4(a-b) presents an example of a simple program and its PDG. (We discuss the figure in more detail below.)

In this paper, we use program slicing to isolate the portion of analysis programs that generates table indices. Specifically, given an input program P we want to extract the statements relevant to the scheduling function $\mathcal{A}$, i.e., those that transform an input i into a set of table indices $Ind(i)$. To do so, we generate a backward slice including statements that affect the value of indices in $Ind(i)$. The resulting slice S will contain a superset of the statements of interest. We then leverage the domain-specific nature of such programs to refine the output of slicing and generate scheduling functions in a fully automated way. To the best of our knowledge this application of program slicing to the domain of IDS parallelization is novel, and an important contribution of our work.

We have developed two algorithms to generate the scheduling function $\mathcal{A}$ via program slicing. The first algorithm, presented in §4.2, is optimized for the common case where the scheduling function $\mathcal{A}$ can be expressed as straight-line code. The second algorithm, presented in §4.3, refines the first to produce better results when S includes conditional instructions.

4.2 Flow-insensitive Algorithm

In §3 we introduced the idea that IDS analyses can be divided in two broad classes: simple analyses whose scope can be expressed in terms of protocol-level units (e.g., analyses aggregating traffic by source address, connection, etc.) and more complex ones with non-trivial scope (e.g., our *multistep* example). We begin by describing an algorithm targeted at the former, simpler class.

Our algorithm is based on the insight that, for many simple analyses, the index used to access analysis state is either an input parameter, or is obtained by a simple, straight-line computation from the input parameters (e.g., extracting a struct field, such as in Figure 1(c)). In these cases the value of the index does not depend on conditional instructions. Therefore, scheduling function generation can be greatly simplified by only considering data dependencies.

Algorithm 1: Flow-insensitive $\mathcal{A}$ generation

```
1  Compute DDG G from program P
2  Find the set of table accesses C in G
3  Compute the backward slice S from C
4  Remove redundant table accesses from S
5  Emit code for S
```

Algorithm 1 lists the high-level steps through which scheduling functions are generated. We describe each step through an example from the *multistep* application introduced in §3.1. The example consists of one of the application's event handlers, *HttpRequest*. Pseudocode for the event handler is given in Figure 4(a). Input parameters are 1) the identifier of the connection generating the request, 2) the request method (e.g., "GET" or "POST"), and 3) the URI being requested. The handler first checks whether the request matches some preconditions, then verifies if an entry for the connection originator exists in its state table. If so, and the detection state associated with the originator is in the WAIT_HTTP state, the handler advances the detection state to WAIT_FTP.

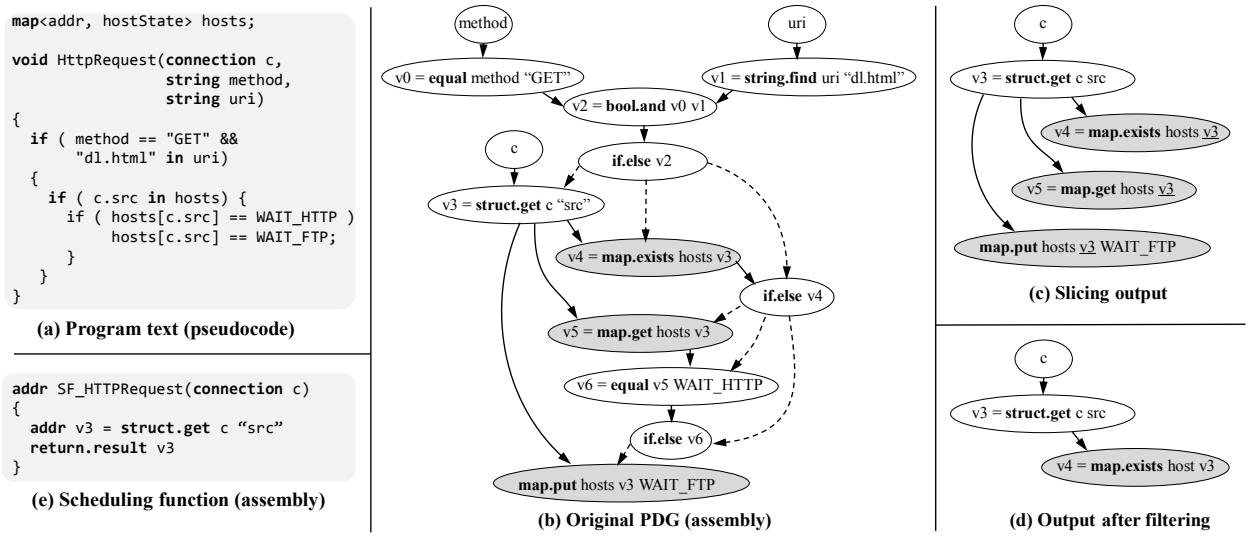


Figure 4: Flow-insensitive algorithm. Dashed arrows represent control dependencies; shaded nodes represent table accesses.

Step 1 in Algorithm 1 computes the PDG. The algorithm only considers data dependencies, so the result is really a data dependency graph (DDG). Figure 4(b) shows the full PDG for the program; dashed lines represent conditional dependencies (ignored in this phase). Note that the original program has been converted to the intermediate assembly-like representation of HILTI ([38]; see §6.2), where each node corresponds to an atomic instruction (conditional statements correspond to *if.else* branch instructions). Step 2 computes the set C of table accesses (shaded nodes in the graph). Step 3 performs backward slicing as described in §4.1, returning the slice S which contains all program statements relevant for the scheduling function $\mathcal{A}$. Figure 4(c) describes the output of this step for our example. Step 4 (Figure 4(d)) filters redundant table accesses, i.e., accesses that use the same index variable such as *v3*, (which corresponds to the high-level variable *c.src* in Figure 4(a)). These are easily identifiable since as part of the PDG construction we transform the code into SSA form [20]. Finally, during code generation (Step 5) the slice is translated to a straight-line code sequence, and each table access is replaced by an instruction returning the corresponding index. The scheduling function $\mathcal{A}$ for our example is reported in Figure 4(e).

Due to the simplicity of the example, the description above does not account for the situation where some index value depends on control flow, or where the application accesses multiple indices. The algorithm transparently deals with these occurrences by returning all possible indices that the execution may generate.

Discussion: We find that the algorithm described in this section works well in a variety of use cases (see §6.4). Its main limitations are that (i) being flow-insensitive, it cannot soundly analyze programs that have loops in the index computation; and (ii) when table indices depend on conditional instructions, the resulting scheduling function is imprecise. In the next section we present a slicing algorithm that overcomes these limitations.

4.3 Flow-sensitive Algorithm

Certain analyses contain conditional constructs, such as branches and loops, in their index computation. These are typically the analyses for which the scope cannot be defined simply in terms of protocol units. One such example is the *ProtocolConfirmation* handler in our *multistep* example, described in §3.1. To be effective, a scheduling function $\mathcal{A}$ for such a program P should keep as much as possible of the original control flow. To do so, the backward slice S on P must include both control and data edges.

We note, however, that even with this approach it is not possible to preserve all control flow. In fact, the slice S may retain branch instructions whose outcome depends on the content of the table, such as for example *<if elem1 in table then v = table[elem2]>*. Such an expression cannot be executed in a scheduling function, which "lives" in the scheduler and does not have access to the table. We deal with this situation by pruning such branch instructions (*<if elem1 in table>*), thus causing the instructions that depend on them (*<v = table[elem2]>*) to be executed unconditionally.

Algorithm 2: Flow-sensitive $\mathcal{A}$ generation

1. Compute PDG *G* from program *P*
2. Find the set of table accesses *C* in *G*
3. Compute the backward slice *S* from *C*
4. Remove from *S* branch conditions that are data-dependent on statements in *C*
5. Remove superfluous branch conditions from *S*
6. Remove redundant table accesses from *S*
7. Recompute the slice *S*
8. Emit code for *S*

Algorithm 2 describes the high-level steps through which flow-sensitive scheduling function generation is performed. We discuss each step using a simplified version of the *ProtocolConfirmation* event handler from *multistep* (§3.1). Pseudocode is given in Figure 5(a). The program receives as inputs a connection ID and a protocol ID. If the protocol is SSH and the destination port is 2222, the handler creates a new entry for the connection responder in its state table. If the protocol is IRC, the handler checks if the table has an entry for the connection originator; if yes, it emits an alert.

Steps 1-3 of Algorithm 2 correspond to the same steps in Algorithm 1, with the difference that in Algorithm 2 the analysis builds the full program dependency graph. Therefore, the slice S returned by Step 3 contains both control and data dependencies (Figure 5(b), with control edges represented as dashed arrows). Steps 4-7 further prune the subgraph. In Figure 5(a) and (b), nodes removed during each step (and the corresponding lines in the high-level pseudocode) are marked with the step number. Step 4 removes branch

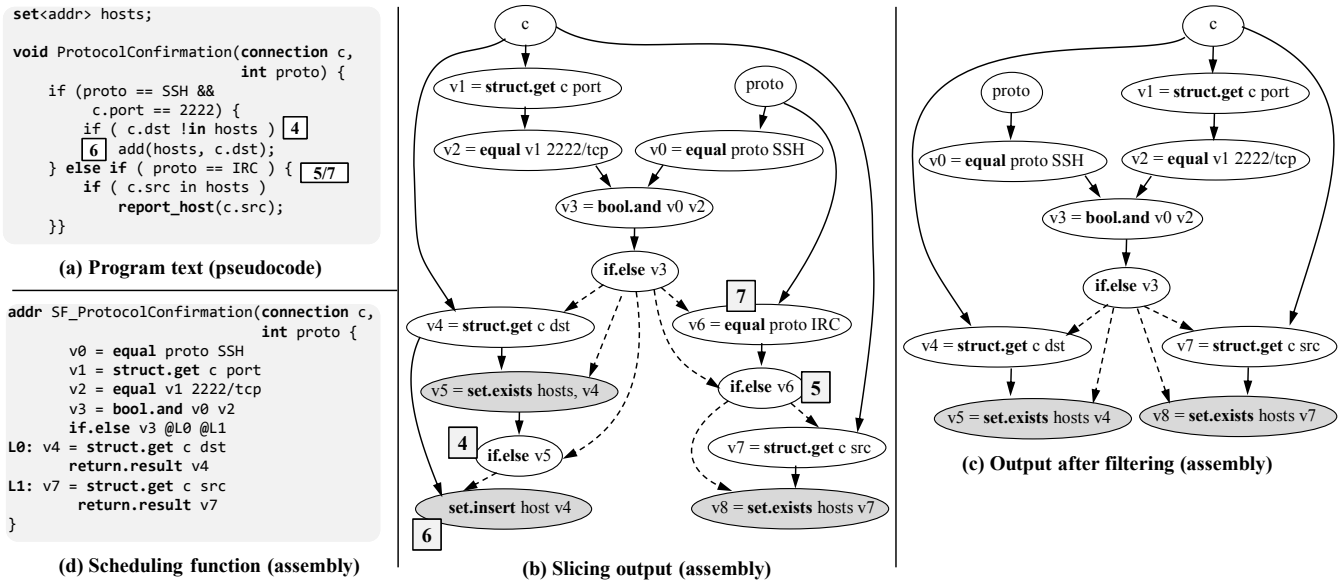


Figure 5: Flow-sensitive algorithm (multistep example)

instructions that cannot be decided at run-time, as discussed above. In Step 5, we also heuristically remove two classes of superfluous branches: redundant ones, i.e., branches that would lead to the same set of indices $Ind(i)$ regardless of whether they are taken or not, and branches for which one side would not lead to any table access. Step 6 removes redundant table accesses, similar to Step 4 in Algorithm 1. Finally, since the pruning performed in Steps 4-6 may have disconnected further nodes from the rest of the graph, in Step 7 the program slice S is recomputed to filter them out. The resulting graph, and the scheduling function emitted by code generation (Step 8) are reported respectively in Figure 5(c) and 5(d).

Discussion: By preserving part of the original control flow Algorithm 2 supports loops within scheduling functions, and can generate more precise results than Algorithm 1.

4.4 Soundness of Scheduling Functions

We define a scheduling function $\mathcal{A}$ to be *sound* if $Ind(i) \subseteq \mathcal{A}(i)$, i.e., if $\mathcal{A}$ returns, for every input i, a superset of the indices the program P will access on that input. A relevant issue is whether the algorithms described in this section generate sound scheduling functions, because this guarantees that each detector instance receives all the inputs relevant for its task. A program P is transformed into a scheduling function $\mathcal{A}$ through two operations: a slicing procedure, that extracts a slice S from P, and the pruning step, that further removes various statements from S and restructures control flow. Intuitively, both transformations must preserve soundness.

We first observe that, as our slicing procedure is correct, it preserves all statements in P relevant to compute $Ind(i)$. Therefore executing S on any input i generates a set of indices $S(i)$ s.t. $Ind(i) \subseteq S(i)$. We then need to show that $S(i) \subseteq \mathcal{A}(i)$.

For a slice S, we define $\pi_S(i)$ as the program path on input i, i.e., the sequence of instructions executed by S when run on input i. We then define the set of all possible paths executed by S as Π_S.

Both Algorithms 1 and 2 create an overapproximation $\mathcal{A}$ of S by removing some (or all) branches and executing the dependent instruction unconditionally. Therefore, for each input i, $\mathcal{A}$ will generate a finite *set* of paths $\Pi_{\mathcal{A}}(i) \subseteq \Pi_S$. The set $\Pi_{\mathcal{A}}(i)$ has the following property. For every input i, let $\pi_S(i)$ be the path generated by the slice S. Then there exists a path $\pi_{\mathcal{A}}(i) \in \Pi_{\mathcal{A}}(i)$, that executes the data-flow instructions in $\pi_S(i)$. Note that if $\pi_{\mathcal{A}}(i)$ executes the same data-flow instructions as $\pi_S(i)$, it will generate the same table indices. Informally, the property implies that for each possible input i, $\mathcal{A}$ generates a superset of the indices returned by S on the same input. Therefore $Ind(i) \subseteq S(i) \subseteq \mathcal{A}(i)$, and $\mathcal{A}$ is sound.

4.5 Running Multiple Scheduling Functions

A full-fledged IDS is expected to run several different analyses on the same traffic. In general, each analysis can have a different scope, and a different scheduling function. When an event is generated, the IDS must therefore run the scheduling functions for all analyses registered for that event. However, the number of possible scopes will be substantially smaller than the number of analyses in the system. Indeed, a 2009 analysis of Bro's script corpus showed that the majority of event handlers could be mapped to one of only four scopes [37]. If two scheduling functions have equal scope only one needs to be executed, reducing the amount of computation required. Furthermore, it may be possible to merge two scheduling functions with different scopes, as long as one *subsumes* the other. For example, the scope *<src IP, dst IP>* subsumes the scope *<src IP, dst IP, src port, dst port, protocol>*. Scheduling according to the former scope is safe even if the application uses the latter, more specific scope.

5. SCHEDULING ALGORITHMS

Scheduling functions can infer scope for each program execution, but they do not imply any concrete scheduling algorithm. An issue is therefore how to perform scheduling efficiently. This section formalizes the scheduling problem as an invariant, and discusses how scheduling algorithms can maintain this invariant.

An IDS analysis within our model can be formalized as a program P that executes in an event-driven fashion, updating its state every time a new input is received. Given such a program, let $\mathcal{I}$ be the space of inputs and $\mathcal{S}$ be the state space of the program. The type of P is $\mathcal{I} \to (\mathcal{S} \to \mathcal{S})$, i.e, given an input $i \in \mathcal{I}$ and a state $s \in \mathcal{S}$ the new state of the program is given by $P(i)(s)$. Without loss of generality, assume that the state space of the program is a

possibly unbounded array or table $T[0 \cdots k]$ of bytes. Given the program P, let $Ind(i)$ be the set of indices of the table T that are read by or written to when executing the program P with input i. Furthermore, we stipulate that P runs within a multi-threaded system, where each execution of P is in general scheduled to a different thread. Each thread maintains its own private copy of the table T holding program P's state. Throughout the rest of the discussion assume that the program P is fixed (i.e., everything implicitly corresponds to the program P).

A scheduler (denoted by Sch) takes a stream of inputs from $\mathcal{I}$ and maps them to threads. Let $m_{Sch}(i)$ be a positive integer that denotes the ID of the thread to which Sch maps the input i. Assume that the scheduler Sch, having already scheduled inputs $i_1, \cdots, i_k$, receives a new input i_{k+1}, and assigns it to the thread ID $m_{Sch}(i_{k+1})$. For a program P we want the scheduler Sch to maintain the following invariant Inv:

$$m_{Sch}(i_x) \neq m_{Sch}(i_y) \Rightarrow Ind(i_x) \cap Ind(i_y) = \emptyset$$

thus enforcing that inputs are mapped to different threads only if the corresponding executions of P do not share state.

5.1 A General Scheduler

To maintain the invariant, a scheduler has to evaluate $Ind(i_1) \cap Ind(i_2)$ for every two inputs i_1 and i_2. In our approach we do not compute $Ind(i)$ directly; rather we assume to have a scheduling function $\mathcal{A}$ such that for all $i \in \mathcal{I}$, $Ind(i) \subseteq \mathcal{A}(i)$.

We will now consider a scheduler $Sch_{\mathcal{A}}$ which makes use of the scheduling function $\mathcal{A}$. Assume that the scheduler $Sch_{\mathcal{A}}$ has already scheduled the inputs $i_1, \cdots, i_k$. Let $m_{\mathcal{A}}(i_j)$ be the thread ID corresponding to the input i_j (for $1 \leq j \leq k$). We assume that the schedule so far satisfies the invariant Inv. Suppose the scheduler receives a new input i_{k+1}. There are three cases:

Case 1: i_{k+1} is equal to i_j for some $j \in [1, \cdots, k]$.
In this case, schedule i_{k+1} on the same thread as i_j.

Case 2: $\mathcal{A}(i_{k+1}) \cap Ind(i_j) = \emptyset$ for all $j \in [1, \cdots, k]$.
In this case, i_{k+1} is scheduled on an arbitrary thread and $m_{\mathcal{A}}(i_{k+1})$ is assigned the ID of this thread.

Case 3: $\mathcal{A}(i_{k+1}) \cap Ind(i_j) \neq \emptyset$ for one or more $j \in [1, \cdots, k]$.
In this case, the set $\mathcal{A}(i_{k+1})$ may overlap with multiple past $Ind(i_j)$, where each i_j was scheduled to a different thread. Therefore, it is in general not possible to pick a thread ID $m_{\mathcal{A}}(i_{k+1})$ that directly maintains the invariant. It is however still possible to ensure that computation remains consistent, by transferring state across threads. For each index $x \in \mathcal{A}(i_{k+1})$, we locate the thread holding the respective table entry $T[x]$. We then consolidate all said table entries on the private state of a single thread. Finally, we schedule i_{k+1} on the same thread, and $m_{\mathcal{A}}$ is updated accordingly.

5.2 Practical Event Scheduling

The scheduler outlined above has two main drawbacks. First, it needs to keep track of past decisions, to ensure consistent scheduling of future events. Also, it may pause computation and move data across threads, in order to ensure that each program run has access to all necessary state. Both issues generate overhead.

We note however that, for many relevant analyses, $Ind(i)$ is a singleton set for all $i \in \mathcal{I}$. In other words, for every input i the program P only reads or writes from a single index in the table T (i.e., $|Ind(i)| = 1$). In particular, this applies to all analyses that do not correlate information across contexts.

As singleton sets cannot partially overlap, case (3) from §5.1 cannot happen, and data movements are never necessary. Moreover, scheduling can be performed statelessly, i.e., without keeping track of past decisions. Let H be a hash function from table indices to positive integers. The scheduler can then simply schedule an input i on the thread with ID $H(Ind(i))$. Since H is a function, the scheduler clearly satisfies the invariant Inv.

In practice, for most relevant detection analyses it is possible to derive a scheduling function $\mathcal{A}$ that returns a singleton set, enabling the use of this simple hash-based scheduler. If a system includes limited number of analyses that *do not satisfy* this condition (i.e., each run of the analysis may access multiple indices), a possible solution is to run them on a dedicated thread/core.

Application	Approach/Key data structures
Flowbytes	Counts the amount of per-flow traffic and generates an event for each flow crossing a given threshold.
Httpvol	Tracks the amount of traffic generated by external HTTP hosts, returns aggregate data sorted by host.
Scandetect	Detects horizontal and vertical port scans, integrating information from connection attempts and HTTP requests.
Multistep [1]	Proof-of concept worm detector, inspired by a didactic policy script for the Bro IDS. Detects worm activity by tracking hosts that generate a specific sequence of events.
Dnstunnel [19]	Simple DNS tunnel detector, tracking hosts that generate a large number of DNS requests without contacting the resolved addresses.
Sidejack [19]	HTTP sidejacking detector. Locates reuses of the same authentication cookie by unrelated users.

Table 1: Summary of applications used in the evaluation.

6. EVALUATION

Experimental highlights. The goal of this section is to evaluate the effectiveness of scope-based scheduling in light of the following questions:

1. *Is our concurrency model effective in exploiting the parallelism present in network traffic?*

   §6.3 shows that our approach is effective in distributing load across multiple CPU cores. Throughput improvements are significant and only limited by the amount of parallelism present in the traffic.

2. *Can scheduling functions be automatically determined via static analysis?*

   The characterization presented in §6.4 shows that our analysis returns scheduling functions that are correct, perform only simple stateless operations, and are close to minimal.

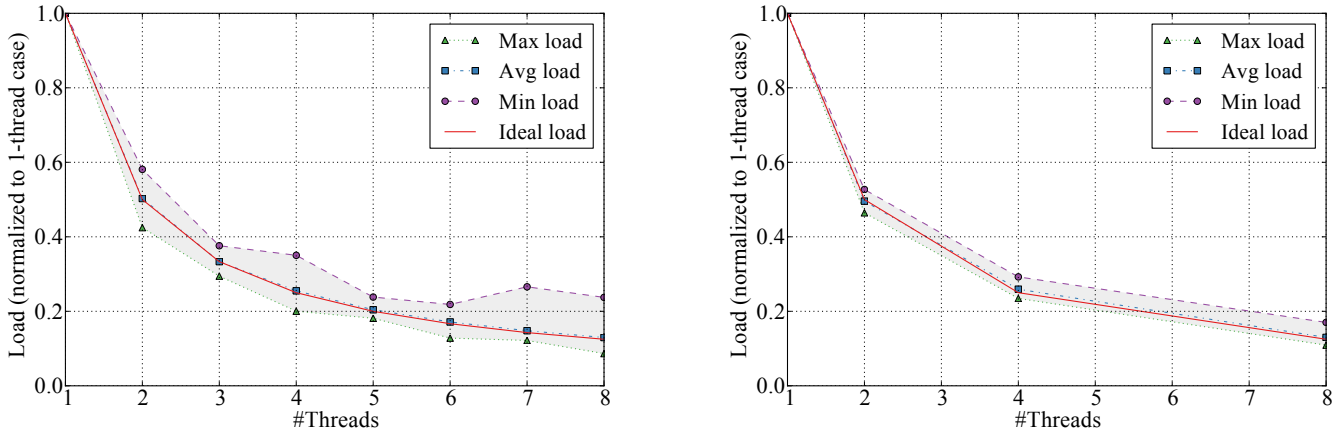
3. *Does our approach bring improvements over traditional multiprogramming techniques?*

   Qualitative analysis in §6.5 suggests that our concurrency model simplifies writing and running efficient IDS analyses, as it transparently provides data isolation.

6.1 Benchmarks and Traces

To perform the evaluation we selected six "intrusion detection kernels", inspired either by existing literature or by discussions with domain experts. Table 1 summarizes them.

These programs represent various classes of operations that IDSs commonly perform; our goal is to evaluate how our technique works on detectors of varying complexity and heterogeneous scopes. In particular, *Flowbytes* and *Httpvol* model measurement scripts that



(a) Full trace *(b) Filtered trace*

Figure 6: Thread load for full and filtered trace

compute common traffic statistics. *Scandetect* is an example of cross-layer analysis. *Multistep* and *Dnstunnel* implement analyses that correlate multiple events/flows to detect suspicious activity. Finally, *Sidejack* is an example of a complex detection heuristic whose scope is defined inside the application layer (HTTP cookie).

We base our evaluation on a packet trace captured at the border gateway of the UC Berkeley campus. In order to obtain a realistic, diverse workload we recorded and merged the traffic flowing through the campus IDS cluster—26 backend systems behind a front-end load-balancer that splits up the total traffic on a per-flow basis [40]—limiting it to the protocols of interest for our applications (HTTP, FTP, SSH, DNS, IRC). The resulting trace comprises 326GB in total volume, includes 349M packets, and covers a timespan of 15 min.

6.2 Analysis/Simulation Framework

One of the design goals of our approach is to be as architecture- and platform-independent as possible. We therefore chose to implement our benchmark in HILTI [38], an assembly-like language designed to be a domain-specific yet abstract representation of traffic analysis tasks. HILTI provides a mix of basic instructions and common high-level primitives (regexp matching, associative containers, etc.), and supports multithreading via lightweight virtual threads. Code generation and execution are achieved using a LLVM-based compiler and a dedicated runtime. In order to carry out our slicing algorithms (§4), we augmented the HILTI toolchain with a simple program analysis infrastructure.

The IDS pipeline outlined in Figure 2 consists of three stages: low-level traffic parsing, event scheduling, and high-level event processing. Currently the HILTI runtime implements all the language features (including multithreading), but lacks traffic parsing and event scheduling functionality. To sidestep this limitation, we emulate the IDS pipeline by implementing the stages separately. Each stage generates intermediate output on disk, which is then fed to the next stage.

In more detail, we first manually generate high-level events by pre-processing the raw network trace with Bro. This distills our network trace into an event trace which includes 298M high-level events. Second, we generate scheduling functions by running our algorithm on the HILTI implementation of our benchmarks. This enables us to verify the correctness and quality of autogenerated scheduling functions (results are discussed in §6.4). Furthermore, we use the scheduling functions to partition the high-level event trace from Stage 1 into sub-traces.

In the third stage, we instantiate a pool of HILTI analysis threads, and we use a trace loader to feed each event sub-trace to a different thread. Each thread performs all the analyses described in Table 1. This setup faithfully reproduces how events would be distributed among threads in a full system, and allows us to evaluate the performance of multiple threads running within our concurrency model. Results are discussed in §6.3. We run all of our experiments on a 64-bit Linux system with two quad-core Intel Xeon 5570 CPUs and 24GB of RAM.

6.3 Parallelism in Network Traffic

This part of our evaluation addresses the issue of whether our approach effectively exploits parallelism present in network traffic. In this experiment, we analyzed load balancing and throughput of the system when running with 1 to 8 hardware threads. We used the approach outlined above, partitioning the Bro event trace across the analysis threads.

Load balancing: For each run, we compute the *load* of each thread, defined as follows. Let N be the number of threads and T_i^N the processing time for the i-th thread in the N-thread setting. The load for the same thread is defined as $L_i^N = \frac{T_i^N}{T_1^1}$. In other words, "load" describes the processing time for a given thread in N-thread setting, normalized to the processing time in the single-threaded setting. In an ideal situation where work is perfectly distributed among threads, the load for each thread i would be $L_i^N = 1/N$. In practice, load deviates from the ideal, for two reasons. The first is that work is never perfectly distributed, resulting in threads whose load is above or below the ideal (i.e., they perform either less or more than their "fair share" of work). The second is that adding more threads increases resource contention, imposing a certain architectural overhead. Architectural overhead results in the average thread load being greater than $1/N$; the gap is expected to increase as N increases (i.e., the overhead becomes more significant as more threads are added).

Figure 6(a) shows the maximum, minimum and average load (relative to the single-threaded case) among all threads for each run. We verified that all threads remain CPU-bound throughout the test. First, we observe that the average load remains close to the ideal for all the measurements. This shows that the architectural overhead imposed by running multiple threads in parallel is limited (average thread load is within 3% of the ideal). The figure however also shows some imbalance in the distribution of load among threads.

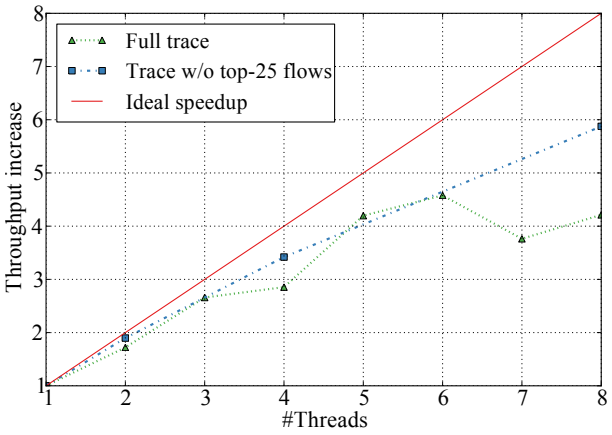


Figure 7: Throughput increase as a function of #CPU cores

Such imbalance can limit the overall throughput, with some threads being overloaded while other ones are not running at full capacity. To explain this result, we investigated the composition of traffic in our trace. Similarly to previous studies [16], we found that the distribution of data among hosts is asymmetric in nature, with the 25 busiest host pairs (of approximately 1M) accounting for 20% of the data. Note that in a system with 8 cores each should carry approximately 12.5% of the traffic processing load to achieve perfect balancing. In our context, even a minor asymmetry in the load distribution can have a significant impact. In order to quantify this impact, we repeated the experiment for the 1-, 2-, 4- and 8-thread cases after filtering out the top 25 host pairs. Results are reported in Figure 6(b). As can be seen, variations in per-thread load are significantly more confined around the average, leading to a more evenly balanced workload. This supports our hypothesis that load asymmetry is chiefly caused by a limited number of "fat flows". We further discuss the issue in §7.

Throughput: For each experiment, we also computed the *throughput increase*, defined as $I^N = \frac{1}{max(L_i^N)}$. I^N quantifies the decrease in load (compared to the single-threaded case) for the busiest thread in the N-thread setting. This definition, albeit simple, has the advantage of allowing to estimate throughput without making assumptions on how the system is implemented. In fact, in absence of buffering and other optimizations, the busiest thread determines the overall throughput. Results, depicted in Figure 7, reflect the conclusion discussed in the context of load balancing. Throughput improvement is significant, although sensitive to asymmetries in load distribution.

6.4 Characterization of Scheduling Functions

We implemented both the flow-insensitive and flow-sensitive algorithms described in §4. They generate the same scheduling functions for all benchmarks, except for *multistep* and *sidejack*, whose slices contain conditional constructs—in this case, the flow-sensitive algorithm produces more precise results. Therefore, in this section we only present results for the *flow-sensitive* algorithm.

Correctness: To ensure correctness of our benchmark implementations, we initially run them sequentially on sample synthetic traces and manually verified their output. The next step was to ensure that the benchmarks keep working correctly when parallelized using the scheduling functions generated by our algorithm. To do so we compared the output of the parallelized and the sequential version on a variety of test traces, verifying their equivalence.

Effectiveness: Figure 8 presents simplified, high-level versions of event handlers and respective scheduling functions from three of our benchmarks for illustration. For most applications, scheduling functions are minimal and perform simple struct field extractions/-concatenations. A few cases present opportunities for further optimization apparent to a human expert. For example, the function returned for *multistep*'s *ProtocolConfirmation* (Figure 8b) performs several checks that are not relevant, since the return value can be determined purely by observing the protocol detected (*proto*).

Overall, results in this section suggest that our approach is effective in producing correct and compact scheduling functions.

6.5 Characterization of Concurrency Model

A relevant question is how our concurrency model fares—in terms of ease of use and efficiency—compared to traditional multiprogramming techniques that make use of inter-thread synchronization. In order to perform a qualitative comparison, we created an alternative implementation of our benchmarks that relies on locks to provide data isolation between threads. In this version, a scheduler distributes events randomly among threads, and analyses use global locks to guard accesses to shared data structures.

When developing the benchmarks for our lock-free concurrency model we were able to develop code in a single-threaded setting, and directly reuse it in a parallel scenario. Conversely, implementing the lock-based version required some modifications to ensure critical sections were properly guarded. In terms of performance, as expected the intensive use of locks led to little observable throughput improvement as more threads were added (limited to 6% on our traces). While switching to lock-free data structures is certainly possible, this would add further complexity to the application and runtime design. Moreover, each handler must at the very least hold logical locks on the individual table element(s) while they are being updated. Conversely, our approach guarantees ordering and minimizes inter-thread synchronization by design, which proves to be a natural fit for programming IDS analyses.

7. DISCUSSION

Scheduler throughput: Scheduling functions are stateless and consist mainly of short, straight-line code sequences, enabling high scheduling throughput. Still, on a large system the scheduler could potentially become a bottleneck, as it executes scheduling functions on all incoming events. We note that many analyses tend to slice traffic using simple scopes, such as per-host or per-connection. The IDS runtime can therefore provide highly optimized, "hardcoded" versions of such scheduling functions. Another solutions to scale further would be to parallelize the scheduler itself [30]. This can however introduce ordering issues, which we discuss in the following. We also note that not all events require scheduling decisions. In particular, timed events—such as state expirations—are implicitly associated with the thread that generated the timer.

Event ordering: If low-level traffic parsing and/or event scheduling are parallelized at the connection level, events generated by different flows may not reach the analyzer in order, as low-level processing of different flows may be performed by different threads. This could affect high-level heuristics whose scope includes multiple connections. A possible solution is to associate each event with a timestamp, and to then use the timestamps to sort the threads' input queues, as in [49]. Another approach is to make the detector agnostic to order, i.e., events are correlated as long as they are received within a certain time interval.

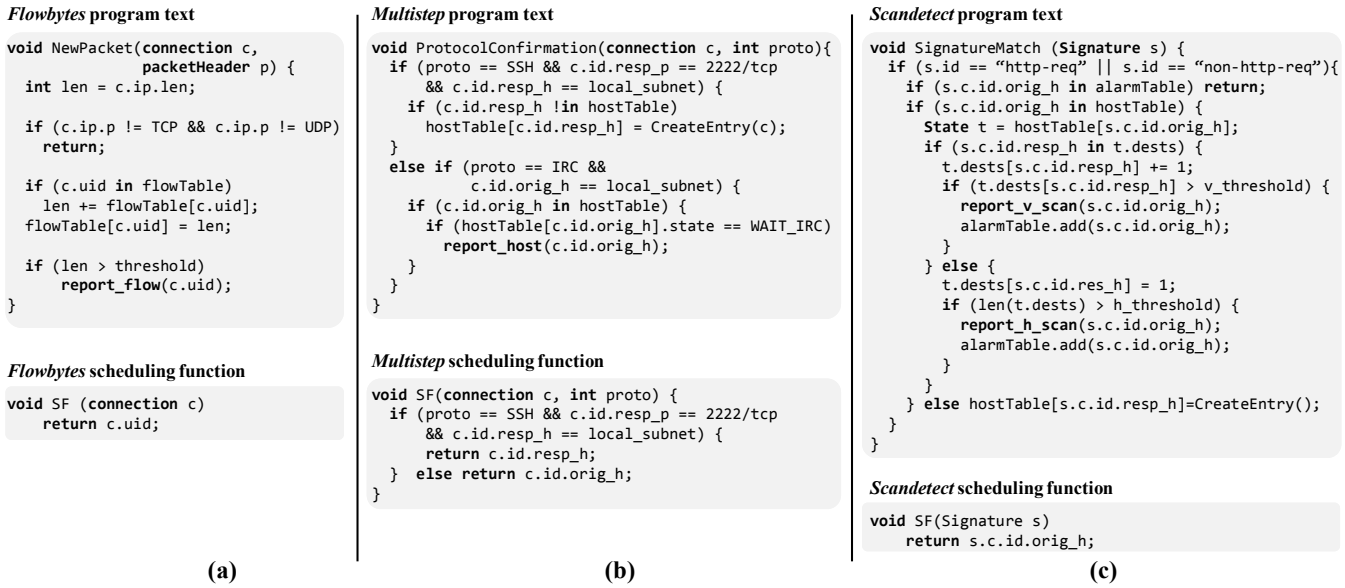


Figure 8: Examples of scheduling functions

Sharing data structures between applications: in our discussion of the scheduling problem, we assumed that each application maintains its private data structures. In practice, separate traffic analyses may collaborate by sharing state. If their scopes are compatible, the analyses can be treated as a single entity for scheduling purposes. If this is not possible, the functionality of one of the analyses could be replicated within the other. The resulting overhead may be acceptable if it allows the analyses to be independent.

Load-balancing: in §6.3 we showed that imperfect load-balancing can limit throughput. We pinpointed the issue to a few high-bandwidth flows accounting for a significant percentage of the total packet stream. We note that this is an intrinsic limitation of hash-based approaches, determined by the amount of parallelism available in the trace. Such traffic would cause load imbalance even for IDSs that perform per-flow pattern matching, since each high-bandwidth flow would still be processed by a single thread. This limitation may be acceptable; indeed hash-based approaches are currently used in deployed systems, such as the NIDS cluster [40]. It should also be noted that often a detector only needs to parse the first few packets of a flow [29]. Therefore, processing of large flows could be halted after a certain byte threshold, preventing imbalance. An alternative solution is to allow flows to be remapped across cores, as in [36].

Customization: The goal of our work is to provide automatic and transparent parallelization of IDS workloads. However, in certain cases the user could want to further optimize the behavior of the system by defining the scheduling functions herself. Enabling this is a matter of the API. In this scenario, our scheduling function generator could be used as a programming aid, providing an initial version of the function that the user could further refine.

8. RELATED WORK

The challenge of IDS parallelization has been examined previously in the literature, with initial efforts focusing on multi-system setups for sharing the load.

[36, 49] focus on efficiently partitioning traffic by flow, and therefore lack the notion of analysis-specific scope. [30] has the notion of a scheduler that separates the traffic into independent subsets based on *event spaces*, manually defined by the IDS operator. The semantics of event spaces are limited in expressiveness, and oriented to statically defining specific contexts (e.g., a specific subnet) more than scopes. Moreover, the authors only consider signature-based detection. [31] discusses an IDS load-balancer that dynamically groups flows based on the similarity of header fields (e.g., source address, port). While this approach is simple, such heuristic correlation is not guaranteed to match the actual detector's scope. Other cluster-based approaches are [23, 40], which are built on a more traditional concurrency model (intra-node synchronization) and, as [30, 31], do not tackle parallelism within multi-core nodes.

There is another body of work (e.g., [28, 41, 43, 44]) on accelerating packet matching on parallel hardware, including GPUs. These approaches are restricted to byte-level pattern matching; while this makes parallelization straightforward—as there is just a single, static scope—it severely limits detection capabilities. In contrast, our work aims to parallelize arbitrary stateful analyses.

The work closest to our discussion is [37], which presents a parallel IDS design for multi-core architectures. The system incorporates the notion of per-handler scope; however, scopes need to be manually defined for each analysis, and they are still limited to sets of protocol header fields. Our work can be seen as an extension of those, presenting a concurrency model which is independent from the specifics of the analysis and automatically derives the parallelization strategy.

Variations of scope-based parallelization have been defined outside the realm of intrusion detection. Our approach is inspired by serialization sets [14], a generic parallel programming paradigm. In serialization sets, each shared object is associated with a *serializer* method, which returns an object-specific key. The runtime uses the key to serialize computations that access the same shared object. Our work adapts this approach to the event-driven paradigm typical of packet processing and contributes efficient scheduling algorithms. Moreover, since our approach is domain-specific to IDSs we can leverage the common structure of IDS programs to compute scheduling functions automatically, without developer interaction.

The networking community has also contributed models aimed at parallel packet processing. [25] describes a parallel stateful packet processing system, where a set of processing blocks are composed in a data flow graph. The system supports a context-ordered mode

where logical blocks can be parallelized by applying serializers (termed *context-designator*) to the input stream. The rigid organization of processing in a pipeline makes the system more suited to traffic processing/shaping than intrusion detection; moreover, the developer is still required to manually specify serializers. [24] outlines an approach to state manipulation, with the goal of simplifying dynamic provision/consolidation of network appliances. State is divided in independent units using *keys*, i.e., combinations of protocol header fields. Our definition of scheduling functions can be seen as a generalization of this approach to state partitioning. [26] focuses on mapping IDS workloads to a set of distributed network nodes. Traffic partitioning is static, similarly to [30], and based on offline workload estimates. [22] presents a parallel software router that optimizes the whole system stack for its specific application, yet does not easily generalize to other types of processing. [21] performs analysis of a pipelined software router using symbolic execution to derive the semantics of each component, while we use static analysis to generate precise executable slices.

The literature also presents a number of general-purpose programming APIs that take different approaches to parallelization (e.g., [3, 4, 17]). Our approach does not strive to be a general layer; instead, limiting the scope to event-driven packet processing enables us to keep the programming model simple and hide concurrency issues from the programmer. Architecture-specific parallel APIs such as CUDA [2] require significant application-specific effort because of their restricted computational paradigm.

Historically, the HPC community has also investigated the problem of compiler-based automatic parallelization. This line of work targets scientific and numerical computations (see for example [13, 15, 32, 33] and Chapter 11 of [12]). Target workloads typically involve repetitive operations over large arrays; therefore, approaches focus on loop vectorization and parallelization. More recently, similar techniques have been proposed for batch processing workloads such as compression, machine learning, and text processing [42, 50]. Our work is similar in spirit, as it strives to exploit domain-specific program features to extract parallelization. However, IDS programs present different requirements (real-time stream processing), for which we leverage different program features (state and computation structured around scopes).

Furthermore, [46] proposes the use of program slicing to partition a sequential program into parallel slices. This approach uses slicing to determine instruction- and task-level parallelism. Conversely, we use it to infer high-level properties of the program (its scope), which enable extensive data-level parallelism.

Finally, Parcae [34] and Varuna [39] optimize parallel execution of multi-threaded programs according to various metrics (time, resource consumption). We see these works as orthogonal, as they could be used to fine-tune the degree of parallelism in our approach.

9. CONCLUSION

Traffic processing presents numerous opportunities for parallelism, but making IDS scalable *and* flexible remains notoriously difficult. In this paper, we propose a domain-specific concurrency model that can support a large class of IDS analyses without being tied to a specific detection strategy. Our technique partitions the stream of network events into subsets that the IDS can process independently and in parallel, while ensuring that each subset contains all events relevant to a detection scenario. Our partitioning scheme is based on the concept of *detection scope*, i.e., the minimum "slice" of traffic that a detector needs to observe in order to perform its function. As this concept has general applicability, our model can support both simple, per-flow detection schemes (e.g., pattern/signature matching) and more complex, high-level detectors. Moreover, we show that it is possible to use program analysis to determine the appropriate traffic partitioning *automatically and at compile-time*, and enforce it at run-time with a specialized scheduler.

Initial results are promising, and show that indeed our approach correctly partitions existing sequential IDS analyses without loss of accuracy, while exploiting the network traffic's inherent concurrency potential for throughput improvements.

Acknowledgments

We thank Drew Davidson, Mohan Dhawan, Aaron Gember-Jacobson, Bill Harris, and Matthias Vallentin for their suggestions, which greatly contributed to the paper. Likewise we thank the anonymous reviewers and our shepherd Michalis Polychronakis.

This work was supported by the US National Science Foundation under grants CNS-0915667, CNS-1228782 and CNS-1228792, and by a grant from the Cisco Research Center. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors or originators, and do not necessarily reflect the views of the sponsors.

10. REFERENCES

[1] Bro hands-on workshop 2009. http://www-old.bro-ids.org/bro-workshop-2009-2/, Feb. 2013.

[2] NVIDIA CUDA. http://www.nvidia.com/object/cuda_home_new.html, Jan. 2013.

[3] OpenMP. http://openmp.org, Jan. 2013.

[4] Threading Building Blocks. http://threadingbuildingblocks.org/, Jan. 2013.

[5] AMD Opteron 6300 series processors. http://www.amd.com/en-us/products/server/6000/6300#, May 2014.

[6] Bro IDS. http://www.bro-ids.org/, May 2014.

[7] Checkpoint security - tales from the crypter. http://www.checkpoint.com/threatcloud-central/articles/2014-01-20-Thwarting-Malware-Obfuscation.html, May 2014.

[8] Errata security: Fun with IDS funtime #3: heartbleed. http://blog.erratasec.com/2014/04/fun-with-ids-funtime-3-heartbleed.html, May 2014.

[9] Intel Xeon processor e5-4657l v2. http://ark.intel.com/products/75290/Intel-Xeon-Processor-E5-4657L-v2-30M-Cache-2_40-GHz, May 2014.

[10] Snort IDS. http://www.snort.org/, May 2014.

[11] Suricata IDS. http://suricata-ids.org/, May 2014.

[12] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[13] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. In *ICS*, 1987.

[14] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP*, 2009.

[15] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Toplas*, 9(4):491–542, 1987.

[16] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[17] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP*, 1995.

[18] S. Bodmer, D. M. Kilger, G. Carpenter, and J. Jones. *Reverse Deception: Organized Cyber Threat Counter-Exploitation*. McGraw-Hill Osborne Media, 1st edition, July 2012.

[19] K. Borders, J. Springer, and M. Burnside. Chimera: a declarative language for streaming network traffic analysis. In *USENIX*, 2012.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Toplas*, 13(4):451–490, 1991.

[21] M. Dobrescu and K. Argyraki. Software dataplane verification. In *NSDI*, 2014.

[22] K. Fall, G. Iannaccone, M. Manesh, S. Ratnasamy, K. Argyraki, M. Dobrescu, and N. Egi. RouteBricks: enabling general purpose network infrastructure. *ACM SIGOPS Operating Systems Review*, 45(1):112–125, 2011.

[23] L. Foschini, A. V. Thapliyal, L. Cavallaro, C. Kruegel, and G. Vigna. A parallel architecture for stateful, high-speed intrusion detection. In *ICISS*, 2008.

[24] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward software-defined middlebox networking. In *HotNets*, 2012.

[25] H. Gill, D. Lin, T. Kothari, and B. T. Loo. Declarative multicore programming of software-based stateful packet processing. In *DAMP*, 2012.

[26] V. Heorhiadi, M. K. Reiter, and V. Sekar. New opportunities for load balancing in network-wide intrusion detection systems. In *CoNEXT*, 2012.

[27] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–60, 1990.

[28] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park. Kargus: a highly-scalable software-based intrusion detection system. In *CCS*, 2012.

[29] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer. Building a time machine for efficient recording and retrieval of high-volume network traffic. In *IMC*, 2005.

[30] C. Kruegel, F. Valeur, G. Vigna, and R. Kemmerer. Stateful intrusion detection for high-speed networks. In *IEEE S&P*, 2002.

[31] A. Le, R. Boutaba, and E. Al-Shaer. Correlation-based load balancing for network intrusion detection and prevention systems. In *SECURECOMM*, 2008.

[32] K. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Rice University, Apr. 1992.

[33] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, Dec. 1986.

[34] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: a system for flexible parallel execution. In *PLDI*, 2012.

[35] T. Reps and G. Rosay. Precise interprocedural chopping. In *SIGSOFT*, 1995.

[36] L. Schaelicke, K. Wheeler, and C. Freeland. SPANIDS: a scalable network intrusion detection loadbalancer. In *Computing Frontiers*, 2005.

[37] R. Sommer, V. Paxson, and N. Weaver. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. *Concurr. Comput. : Pract. Exper.*, 21(10):1255–1279, July 2009.

[38] R. Sommer, M. Vallentin, L. De Carli, and V. Paxson. HILTI: An abstract execution environment for deep, stateful network traffic analysis. In *IMC*, 2014.

[39] S. Sridharan, G. Gupta, and G. S. Sohi. Adaptive, efficient, parallel execution of parallel programs. In *PLDI*, 2014.

[40] 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 *RAID*, 2007.

[41] J. van Lunteren and A. Guanella. Hardware-accelerated regular expression matching at multiple tens of gb/s. In *INFOCOM*, 2012.

[42] H. Vandierendonck, S. Rul, and K. De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *PACT*, 2010.

[43] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High performance network intrusion detection using graphics processors. In *RAID*, 2008.

[44] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: a multi-parallel intrusion detection architecture. In *CCS*, 2011.

[45] J. Verdu, M. Nemirovsky, and M. Valero. MultiLayer processing - an execution model for parallel stateful packet processing. In *ANCS*, 2008.

[46] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T.-f. Ngai, and J. Fang. Dynamic parallelization of single-threaded binary programs using speculative slicing. In *ICS*, 2009.

[47] M. Weiser. Program slicing. In *ICSE*, 1981.

[48] B. Wun, P. Crowley, and A. Raghunth. Parallelization of snort on a multi-core platform. In *ANCS*, 2009.

[49] K. Xinidis, I. Charitakis, S. Antonatos, K. G. Anagnostakis, and E. P. Markatos. An active splitter architecture for intrusion detection and prevention. *IEEE Trans. Dependable Secur. Comput.*, 3(1):31–44, Jan. 2006.

[50] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *HPCA*, 2008.