Advancing Protocol Diversity in Network Security Monitoring

A Refined Software Architecture and Implementation for Efficient Modular Packet-Level Analysis

Jan Grashöfer[1], Peter Oettig[2], Robin Sommer[3], Tim Wojtulewicz[3], and Hannes Hartenstein[1]

[1]Karlsruhe Institute of Technology, KASTEL – Institute of Information Security and Dependability
{jan.grashoefer,hannes.hartenstein}@kit.edu
[2]Karlsruhe Institute of Technology, Steinbuch Centre for Computing (SCC)
peter.oettig@kit.edu
[3]Corelight, Inc.
{robin,tim}@corelight.com

Abstract

With information technology entering new fields and levels of deployment, e.g., in areas of energy, mobility, and production, network security monitoring needs to be able to cope with those environments and their evolution. However, state-of-the-art Network Security Monitors (NSMs) typically lack the necessary flexibility to handle the diversity of the packet-oriented layers below the abstraction of TCP/IP connections. In this work, we advance the software architecture of a network security monitor to facilitate the flexible integration of lower-layer protocol dissectors while maintaining required performance levels. We proceed in three steps: First, we identify the challenges for modular packet-level analysis, present a refined NSM architecture to address them and specify requirements for its implementation. Second, we evaluate the performance of data structures to be used for protocol dispatching, implement the proposed design into the popular open-source NSM Zeek and assess its impact on the monitor performance. Our experiments show that hash-based data structures for dispatching introduce a significant overhead while array-based approaches qualify for practical application. Finally, we demonstrate the benefits of the proposed architecture and implementation by migrating Zeek's previously hard-coded stack of link and internet layer protocols to the new interface. Furthermore, we implement dissectors for non-IP based industrial communication protocols and leverage them to realize attack detection strategies from recent applied research. We integrate the proposed architecture into the Zeek open-source project and publish the implementation to support the scientific community as well as practitioners, promoting the transfer of research into practice.

1 Introduction

In network security monitoring, stateful deep packet inspection is used to passively obtain detailed information about the communication in a network. This information allows operators to unveil direct and indirect security threats, such as intrusion attempts and misconfigurations. Network Security Monitors (NSMs) log summaries of their observations for forensic purposes, generate alerts or trigger instant countermeasures upon detecting malicious behavior [24].

As information technology spreads into ever new domains, trends like the Industrial Internet of Things (IIoT) lead to a diversification of the traditional TCP/IP-based protocol stack [7, 46]. For example, numerous works that address security in production [10, 47, 45, 32] or energy systems [25, 4] show the large demand for deep packet inspection (DPI) of non-IP protocol stacks. This need is also unabated with regard to the proliferation of machine learning, as it has been shown that the knowledge of protocol semantics has a higher influence on the quality of trained models than the applied algorithm [3]. However, existing research work that extends established open source monitoring software like Snort, Suricata or Zeek/Bro to support new low-level protocols, is hardly ever transferred into practice[1]. The integration of new dissection capabilities is severely hindered by the far-reaching changes these extensions require to the monitoring tools, because the low-layer processing, which operates at the granularity of packets, is typically hard-coded, due to performance concerns. Consequently, established monitoring solutions are limited to a narrow, TCP/IP-based protocol stack.

[1]For example, the work of Kabir-Querrec [25]: https://github.com/zeek/zeek/pull/76

To advance protocol diversity in network security monitoring, we develop a refined NSM architecture for efficient modular packet-level analysis. Modular protocol dissectors have three main advantages: 1) Separating the core monitor functionality, i.e. the orchestration of the analysis, from the protocol dissection significantly simplifies the maintenance of the monitor core as well as the implementation of new dissectors for external developers. 2) Providing dissectors as independent modules using well-defined interfaces facilitates their reuse in new contexts. 3) Modularity allows NSM operators to tailor the monitoring system to their deployment-specific needs. As a monitor is confronted with arbitrary input that might even be crafted to attack the NSM itself [35, 31, 37], the increased flexibility aids in minimizing the potential attack surface of the monitor.

Given the inherent complexity of dissector development and the ever growing number of protocols, established tools already introduced plugin interfaces to decouple the domain-specific development of dissectors for application-layer protocols. Protocol dissectors for protocols that build upon TCP/IP are able to exploit the existing interfaces. Examples like DNP3 [28] or Modbus dissectors are widely adopted, reused, and are even being continuously developed [22, 9, 42]. However, the existing plugin interfaces cannot be easily extended to support packet-level dissectors, as these interfaces focus on a different level of abstraction. In this work, we introduce a refined software architecture for NSMs that addresses the unique challenges of packet-level analysis. Among these, performance represents the key challenge for a dynamic plugin scheme compared to the previous, hard-coded processing at packet-level. Continuous monitoring for applications like intrusion detection requires online-processing to guarantee timely reactions, in spite of high traffic volumes or resource-constrained monitoring devices. For modular packet-level analysis, dispatching performance becomes crucial, because dispatching of packet-level dissectors is done for *every* layer of *every* packet.

So far, the lack of protocol support beyond the traditional Ethernet-based TCP/IP stack prevents the application of established network security monitoring technologies to protect sensitive resources like critical infrastructures. By developing a refined network security monitor architecture for efficient modular packet-level analysis, we seek to close this gap. Our contributions can be summarized as follows:

- We identify the challenges for modular packet-level analysis in DPI, present an extended NSM architecture to address them, and derive requirements for its implementation.
- We evaluate the performance of data structures for dissector dispatching and show that array-based approaches outperform hash-based data structures in this case, while arrays can keep up with hard-coded packet-level processing.
- We implement the proposed architecture into the established Zeek network monitor, modularize its previously hard-coded packet-layer stack, and measure a negligible performance impact on the overall monitor performance. Furthermore, we simplify dissector development by supporting the use of a tightly integrated parser generator, which allows for adding new dissectors without writing any C++ code.
- We demonstrate the utility of the extended architecture on the example of dissectors for industrial communication protocols, namely GOOSE and ProfinetIO, by leveraging them to realize basic, domain-specific attack detection strategies as proposed in recent research.

The overall goal of this work is to allow for the application of established network security monitoring technologies in new domains and, in particular, to support the transfer of current research into practice. In addition to providing the discussed examples as open source, our implementation of the extended NSM architecture has been released as part of Zeek 4.0.

The rest of the paper is structured as follows: First, we discuss related work in Section 2. Then, we provide background on NSM architecture in Section 3. In Section 4, we propose an extended NSM architecture for modular packet-level analysis, specify requirements for its implementation, and present our implementation in the open-source NSM Zeek. In Section 5, we evaluate the performance of data structures for dissector dispatching and assess the performance impact of the extended architecture on the overall monitor performance. Finally, we showcase a set of practical applications to demonstrate the utility of a refined architecture for modular packet-level analysis in Section 6. This covers the migration of Zeek's packet-level protocol stack, the integration with the Spicy parser generator toolchain [41], the implementation and exemplary application of dissectors for ICS protocols, as well as an additional feature to keep track of unknown protocols. Section 7 concludes the paper.

2 Related Work

The established open source security monitoring tools, Snort, Suricata, and Zeek/Bro, support a modular interface for dissectors based on TCP or UDP. However, only Snort offers interfaces to customize processing of lower layers. Snort 2 supports so-called *preprocessors* for the reassembly of streams and additional detection capabilities [26]. While this allows to process other protocols, the integration with other components like the rule engine is very limited. Preprocessors also ignore layering so that every packet is processed by every preprocessor. Snort 3 refines preprocessing and introduces a maximum of 256 modular *codecs* that integrate

into the processing pipeline focusing decapsulation and allow "basic per-frame validation" [34]. Internally, Snort 3 maintains its own mapping of identifiers to codecs that is partially hard-coded. This approach requires the codecs to maintain an additional mapping between the protocol-specific identifiers and the ones used by Snort. Furthermore, the selection of Snort-specific identifiers needs coordination between extension developers to prevent the duplicate use of an identifier.

The need for modularization of network monitoring was also recognized by Casola et al. [7]. They extend a commercial monitoring software to support modular protocol dissectors focusing on IoT and wireless sensor networks. They implement the monitoring of the IPv6 over Low power Wireless Personal Area network (6LoWPAN) protocol. While Casola et al. implement a modular system, they neither provide details about their interface design, nor evaluate the performance impact introduced by modularization. In fact, the presented architecture uses distributed probes that capture the traffic, encapsulate each packet, and forward the traffic to the actual monitor. This suggests that the dissectors for network layer protocols are implemented on top of the existing IP analysis stack. The approach to encapsulate low-layer protocols in IP-based protocols to work around the limitations of the existing plugin mechanisms is also found in other work. For example, the Idaho National Laboratory (INL) released a protocol dissector that requires a converter to support an industrial automation protocol, which is primarily specified for serial communication [11].

The analysis of protocol headers is also a performance critical aspect for networking hardware, because the headers encode information that network components like switches, routers, and firewalls rely on to make their forwarding decisions. In their work “Design principles for packet parsers”, Gibb et al. review the process of dissecting multilayered packets in hardware to obtain relevant data like addresses [19]. The authors discuss general challenges of packet parsing, describe the notion of parse graphs as state machines, and define an abstract model of packet parsers. Furthermore, Gibb et al. consider reconfigurability. However, their work addresses a different domain by focusing on high-speed packet processing in hardware using ASICs. With the advent of software defined networking, flexibility in terms of parsing low-level protocols has become a major concern. Bosshart et al. introduced a domain-specific language for Programming Protocol-independent Packet Processors (P4) [5] that gained significant momentum. P4 fosters hardware independent reconfiguration of network devices in the field to support new header formats. While these works underline the need to support protocol diversity, they do not extend to network security monitoring, which focuses on the comprehensive reconstruction of the observed communication up to the application layer.

3 Background on Network Security Monitor Architecture

A network security monitor faces two fundamental challenges to provide visibility into network communication: First, descriptive information must be extracted from the network traffic using deep packet inspection. Second, the extracted information must be processed with respect to operator-defined network policies (e.g., to raise alarms) or persisted for retrospective analysis (e.g., in the form of logs). In this work, we focus on the first aspect of extracting information from the observed traffic. In general, the processing of traffic in a network security monitor can be divided into three levels of abstraction (c.f. [27]): Packet-level, session-level, and artifact-level. At the **packet-level**, the monitor extracts information that is defined at the scope of a single packet such as Ethernet addresses. Packets group information that is sent from A to B and meta data that is required to do so. In this work, we focus on the packet-level of DPI. At the **session-level**, extracted information is combined to identify sessions (e.g., 5-tuple for a TCP/IP connection) that maintain a state. Sessions can be nested; for example, multiple HTTP sessions might reuse a persistent TCP/IP connection. At the **artifact-level**, artifacts like files are reconstructed from potentially multiple sessions. These levels of abstraction are reflected in the software architecture of a network security monitor.

For our work, we define a network security monitor reference architecture as shown in Figure 1. We model our reference architecture based on Zeek/Bro [31], as it proves to be sufficiently general to represent a variety of systems [1, 12, 27]. The key idea is to separate the DPI mechanism from the specification of policies by transforming the ingested network traffic into a stream of high-level events that describe the observed communication. The monitor core is responsible for performing deep packet inspection to generate the event stream. Users interact with the monitor core by specifying policies, providing configuration, and deploying plugins. The user-defined policies act on the stream of high-level events. Examples of events may be the establishment of a TCP connection or the observation of an HTTP header. Depending on the capabilities of the employed policy language, policies may be used for basic inspection and filtering (e.g., to control logging) up to the realization of attack detection logic by correlating multiple events. The configuration as well as plugins can be used to customize the DPI process itself. Using well-defined interfaces (represented as notches in Figure 1), plugins allow for the implementation of additional functionality separately from the monitor core, which simplifies both, the development process of plugins and the maintenance of the monitor core. As plugins can be developed, compiled, and deployed without interfering with the NSM’s code base, NSM developer, plugin developer, and NSM user become in-

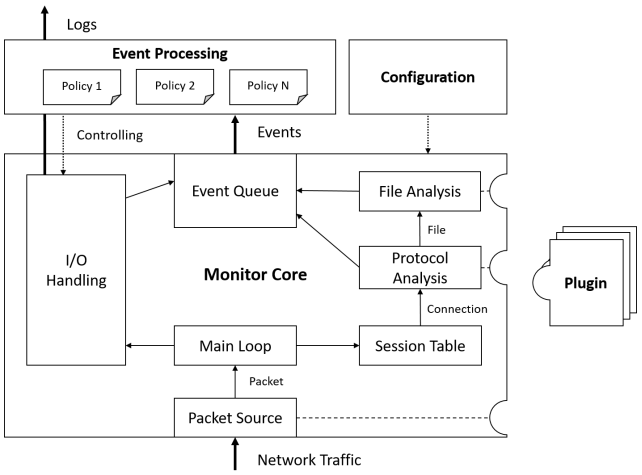


Figure 1: Network Monitor Reference Architecture[2]– The monitor core transforms network traffic into a stream of high-level events that are processed based on user-defined policies, e.g., to control logging. The analysis performed by the monitor core can be customized and enriched by configuration and independently deployable plugins, respectively.

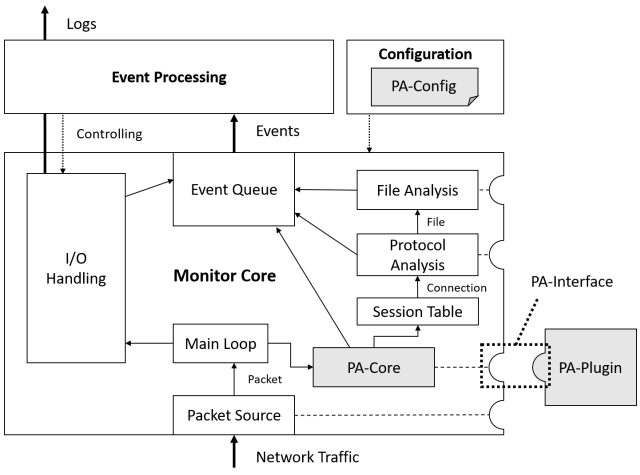


Figure 2: Proposed extended Network Security Monitor Architecture – Packet-level analysis becomes a separate step and provides an interface for plugins. The configuration allows for orchestration of the plugins.

dependent roles. We refer to the users of an NSM as operators.

In the reference architecture, the core's main loop is the central coordinator of the control flow in the monitor and steers the processing of packets and events. Assume the monitor receives a packet that is part of an HTTP connection over TCP/IP. The main loop reads the packet from a packet source. Packet sources abstract different forms of packet ingestion, such as reading from network interfaces or trace files, and are implemented as plugins. A packet source is solely responsible for feeding the raw traffic into the NSM and does not involve parsing of the packet contents. Once our example packet is read from the packet source, the lower layers are processed to assemble the 5-tuple (source address and port, destination address and port, transport protocol) that is used to look up the corresponding TCP connection in the session table. The session table maintains the state of active connections. When the packet is assigned to a connection, the analysis continues on the application-protocol-level, HTTP in our example. If the protocol carries files, further file analysis may be conducted, such as calculating file hashes. Both protocol analysis and file analysis can emit events that are scheduled in the event queue. Finally, the scheduled events are processed with respect to the user-defined policies. Policies may control the handling of in- and output such as blocklists and logs as well as implement correlation and detection logic.

While the current architecture allows for easy extension of the monitor core by additional application-level protocol dissectors and file analyzers using the available plugin interfaces, the processing of packet-level protocols is typically hard-coded. In the following, we analyze the process of packet-level analysis and propose an extended architecture for modular packet-level analysis.

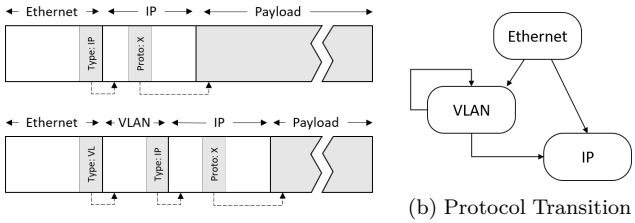
[2]based on "Following the Packets: A Walk Through Bro's Processing Pipeline" [40]

4 Design and Implementation of Modular Packet-Level Analysis

In this section, we propose an extended NSM architecture for packet-level analysis, describe the unique challenges at the packet-level, and derive the corresponding requirements for implementation. Finally, we present our implementation of a framework for modular packet-level analysis in Zeek. In section 5, we evaluate the related performance aspects.

4.1 Proposed Extended Architecture and Problem Analysis

With respect to the reference NSM architecture as described in Section 3, modular packet-level analysis means to split the packet analysis code by protocol, decouple protocol-specific dissectors from the monitor core by employing the plugin mechanism that facilitates extensibility, and enable operators to orchestrate the interaction between the dissectors. Figure 2 shows our proposal for an extended network monitor architecture. We introduce packet analysis as a separate processing step (PA-Core) in the monitor core. The core part of our framework serves as the entry point for packet-level analysis and defines the interface (PA-Interface) that is implemented by packet analysis plugins (PA-Plugin). The PA-Interface defines how the PA-Core interacts with PA-Plugins to perform packet processing, while



(a) Example Packet Structures

(b) Protocol Transition Graph

Figure 3: Illustrative example of nested packet structures and a corresponding protocol transition graph inspired by [19].

the PA-Plugins realize the protocol dissection. To allow for orchestration of the analysis process by operators, we introduce the packet analysis configuration (PA-Config).

The actual challenge for the implementation of the extended architecture lies in the diversity and the nested structure of protocols in today's and future's networks on the one hand and on the required performance of packet-level analysis on the other hand. Let us recall that packet-level protocols are combined by encapsulating their Protocol Data Units (PDUs) into each other. A PDU consists of its payload, i.e., data to be transferred, and a header that contains meta data required to do so. Encapsulating protocol Y into X means that Y's PDU becomes the payload of X's PDU, so that the resulting packets consist of several layers represented by their protocol headers. Figure 3a shows two exemplary packet structures that carry IP over Ethernet with one of the packets using an 802.1Q VLAN tag. In real-world environments, packets may consist of eight layers and more [19].

On the packet-level, protocol headers typically contain a numerical value that identifies the encapsulated protocol. The process of determining the encapsulated protocol and passing on the payload (i.e., the encapsulated PDU) for appropriate analysis, we call **packet-level dispatching**. The mapping of protocol identifier to encapsulated protocol is part of a protocol's specification and, thus, might vary between protocols, which makes dispatching context dependent. Note that protocol identification on session-level is more complex [14, 20]. However, protocol identification on session-level is usually done once per session, whereas packet-level dispatching is required for *every* layer of *every* packet. Thus, providing fast and context-aware dispatching represents the core challenge for implementing modular packet-level analysis. In our extended architecture, the PA-Interface codifies how dispatching among PA-Plugins is realized.

In addition to the protocol formats, their interrelations need to be considered for dispatching, i.e. which protocols can be encapsulated in a PDU. Adapted from parse graphs defined by Gibb et al. [19], we define Protocol Transition Graphs (PTGs) as digraphs for which nodes represent protocols and edges depict the possible encapsulation relationships. Figure 3b shows the PTG for the packets shown in 3a. The graph contains a loop for the VLAN protocol to take into account stacked VLAN tags (QinQ, IEEE 802.1ad). Parsing a packet layer by layer can be understood as executing the state machine that is described by the PTG. The current state corresponds to the protocol of the currently analyzed layer. State transitions are executed based on the identifier for the encapsulated protocol that is extracted from the current header. Given the vast amount of protocols and the lack of a comprehensive registry[3] it is neither feasible nor desirable to support monitoring all of them. For example, any dissector that is not required in a particular deployment context unnecessarily increases the attack surface of the monitor. Thus, operators also need the flexibility to adapt the packet-level analysis to their environment. Customizing the analysis means to define the PTG state machine by selecting the relevant protocols and specifying the corresponding transitions. We ensure this flexibility by splitting the analysis task into protocol-specific PA-Plugins and moving the specification of the transitions into a separated PA-Config that can be adapted by the operator.

To summarize the problem addressed in this paper, given the extended network monitor architecture of Figure 2 and the interrelations of the protocols to be analyzed on packet-level as sketched in 3b, one has to define the architectural elements, particularly the PA-Interface, such that the performance penalty of the introduced additional functionality is minimal. This problem, together with more detailed requirements, is addressed in the following.

4.2 Requirements and Implementation

Following a structured approach, we decompose the problem of realizing modular packet-level analysis by deducing detailed requirements. Subsequently, we present our implementation of the previously proposed extended architecture in Zeek, and show that our implementation fulfills the defined functional requirements.

Requirements. We define the following fundamental requirements for implementing flexible packet-level analysis:

R1 – Extensibility of the NSM represents the key requirement and is enabled by modularity. While modularity realizes decoupling of functionalities, extensibility allows to exploit the modularity at deployment time: The addition of new protocol dissectors to an NSM deployment should be possible without modification of the monitor core. Once dissectors can be developed and compiled independently from the NSM core, the process of implementing new dissectors is vastly simplified while

[3] For example, the IANA lists more than 200 assignments for EtherTypes [16]. The list is not comprehensive because even a number of widely deployed protocols such as ProfinetIO, EtherCAT ore GOOSE are not officially registered.

at the same time, the maintainability of the monitor core is improved.

R2 – Universality requires that the PA-Interface accounts for variability with respect to the location of protocol identifiers in the protocol header. For example, assuming to find a protocol identifier at a fixed offset is not viable, because the header structure might even vary within a protocol. One prominent example is IPv6, which uses the last of possibly multiple extension headers to identify the encapsulated protocol [13].

R3 – Independency requires PA-Plugins to be able to schedule events in the event queue, so that policies can be specified referring to packet-level information. This is particularly important for protocols that do not establish sessions but carry relevant information like the Address Resolution Protocol (ARP) [33] or protocols for industrial control systems.

R4 – Configurability allows the operator of the NSM to select the set of analyzers and configure protocol identifier mappings without touching code. On the one hand, this allows the operator to tailor the monitor to its environment and thus reduce the attack surface that the monitor itself offers. On the other hand, configurability accounts for incomplete information. For example, MPLS may not explicitly specify which protocol follows after a label [36] so that the encapsulated protocol has to be inferred based on the deployment context. With respect to the design of the PA-Config, there are two further requirements to consider:

R4.1 – Unambiguousness refers to the fact that each protocol is free to define its own mapping between identifiers and encapsulated protocols. Thus, the same identifier might refer to different encapsulated protocols depending on the context. Conversely, there might be different identifiers for a single protocol. For example, the Ethertype mapping used in the IEEE 802 standard family [15] and the Point-to-Point (PPP) protocol field mapping [39] exhibit both, overlapping and multiple identifiers for the same protocol.

R4.2 – Integrability ensures that the PA-Config allows for chaining dissectors arbitrarily to handle encapsulation. In terms of a PTG this means that it is possible to establish arbitrary loops[4]. For example, data link layer PDUs might be encountered as payloads on higher layers with respect to tunneling protocols. Furthermore, integrability ensures that existing dissectors or chains of dissectors can be reused.

R5 – In addition to the previously described functional requirements, **Performance** is paramount. In case of network security monitoring, losing packets means to irretrievably lose potentially critical observations. Thus performance is ultimately correlated to the visibility a monitor aims to provide. Because nested layers trigger multiple packet-level dissectors for every packet that is seen on the wire, the performance of the packet-level dispatching mechanism becomes a relevant factor. We evaluate the dispatching performance in section 5. Note that apart from monitoring high volume traffic, performance is also crucial with respect to the use of resource-constrained monitoring devices. In the following, we present our implementation of the extended NSM architecture in Zeek and show that it satisfies the functional requirements.

[4]Note that we assume each analysis step to consume a packet layer. Thus, loops in the PTG cannot be exploited to block the progress of the analysis.

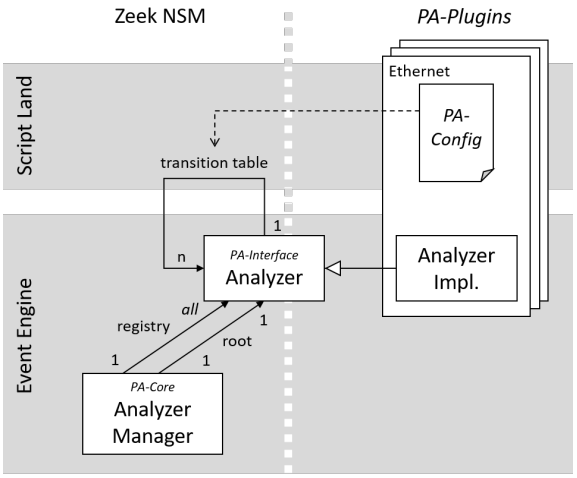


Figure 4: Implementation of the extended NSM architecture for modular packet-level analysis in Zeek.

Implementation. The key idea for the implementation of the extended NSM architecture is to emulate the state machine that is described by a Protocol Transition Graph, where parsing a packet layer by layer can be understood as executing the PTG state machine as described in Section 3. Figure 4 shows an overview of our implementation in Zeek. The monitor core is realized by Zeek's so-called event engine, which is responsible for generating the high-level event stream. Building upon the event engine, Zeek comes with a turing-complete, domain-specific scripting language that acts as the primary user interface for NSM operators. Scripts are used to express monitoring policies as well as to provide configuration for the event engine. To allow for extension of the event engine, Zeek offers a plugin mechanism based on shared libraries.

We define the PA-Interface in form of an abstract dissector super class, called *analyzer*. Derived packet analyzer implementations for a given protocol are provided by PA-Plugins. Each analyzer implementation corresponds to a state of the PTG state machine. The state machine's transition function is split across the different analyzers and represented in form of a transition table per analyzer. Together with an analyzer implementation, each PA-Plugin contains a corresponding PA-Config that specifies the transition tables using a policy script (see 6.1 for an example). The analyzer implementation specifies the actual dissection logic. Typical steps include the verification of the PDU header and

Data Structure		Description	Implementation
Array	Static	Array indexed by the protocol identifier	C++ built-in
	Dynamic	Array of IDs trimmed at front and end	`std::vector`
Tree	Tree Map	Red/black tree	`std::map`
	Array Tree	Tree grouping adjacent identifiers in arrays	custom
Hash Map	Separate Chaining	Handle collisions using a list of items per index	`std::unordered_map`
	Cuckoo Hashing	Calculate multiple candidate indices for an element using independent hash functions	open source [6]
	Universal Hashing	Find a collision free hash function for a given input set from a family of hash functions	based on [44]
	Perfect Hashing	Construct a collision free hash function	based on [18, 21]

Table 1: Data structures evaluated for packet dispatching.

the extraction of the identifier that determines the encapsulated protocol. Further processing might include scheduling events. Finally, the current PDU's payload is forwarded based on the extracted protocol identifier using the transition table. If no suitable analyzer is found, the failed dispatching is logged. Otherwise, the analysis continues with the next analyzer. Both evaluating the PA-Config and the dispatching mechanism are implemented as part of the analyzer super class, i.e. the PA-Interface, as this functionality is shared between all analyzers. The PA-Core is represented by the analyzer manager, which maintains a registry of all available packet-level analyzers as well as a dedicated root analyzer. In contrast to application-level analyzers that are spawned per connection, each packet-level analyzer implementation is only instantiated once by the manager, since on-demand creation would severely degrade performance. The dedicated root analyzer serves as the entry point for the packet-level analysis in the event engine.

The described implementation achieves extensibility (R1) by splitting the analysis task per protocol and moving it into separately deployable PA-Plugins. By placing the responsibility for identifier extraction on the analyzer, the implementation also meets the universality (R2) requirement, as we do not introduce restricting assumptions on the identifier placement. Due to the modular design of the employed script interpreter, PA-Plugins are able to define and emit custom events without the need to modify the NSM core. Hence, the design satisfies the independency (R3) requirement as well. Configurability (R4) is realized by placing the transition table definition into Zeek scripts that can be adapted by NSM operators to customize the packet-level analysis process. The unambiguousness (R4.1) and integrability (R4.2) requirements for the configuration are both addressed by the state-machine-driven approach for dispatching. As each packet analyzer (i.e., state) maintains its own transition table, identifier conflicts between different protocols are prevented. However, this comes at the cost of storing multiple, potentially large transition tables. The integrability requirement (R4.2) is satisfied, since there are no restrictions on possible transitions. For example, a new IP-carrying stack of data link protocols can be added, without the need to reimplement any IP-related functionality. Due to the paramount importance of performance (R5), we examine this aspect separately in Section 5, including an evaluation of the design's memory overhead. All in all, the presented implementation establishes a clear separation between the roles of NSM operators, NSM core developers, and the developers of packet-level analyzers. The implementation is publicly available being merged into the BSD-licensed open source Zeek project[5] and is part of its 4.0 release.

5 Performance Evaluation

This section is dedicated to the performance of the modular packet-level analysis approach as presented in the previous section. During packet processing, dispatching requires to look up the analyzer for a given identifier and to forward payloads accordingly. One of the main concerns is whether it is feasible to replace the hard-coded dispatching with a dynamic plugin scheme on such a critical data path. Note that, for example, just a single download at the speed of 10 MB/s, with each packet consisting of 5 packet-level layers on average, yields more than ten thousand lookups per second. Thus, the dispatching data structure becomes mission-critical. Our goal is to assess and minimize the potentially introduced overhead. In this section, we first evaluate different data structures in the context of the dispatching use case. In a second step, we investigate the impact of the proposed architecture on the overall performance of the monitor, employing the best-performing data structure for our implementation in Zeek. For reproducibility, we release our experiments and code as open-source[6].

5.1 Dispatching Data Structures

For dispatching, we require a simple data structure that maps protocol identifiers to packet analyzer instances.

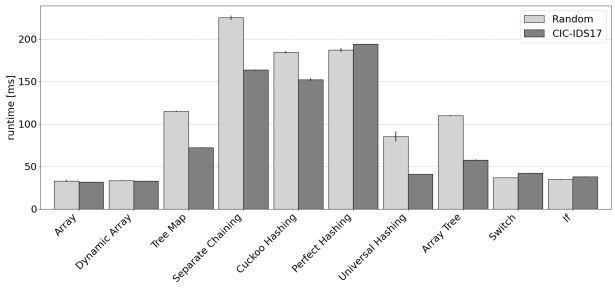
[5] https://github.com/zeek/zeek
[6] https://github.com/kit-dsn/packet-analyzer-benchmarks

Table 1 provides an overview on the evaluated data structures. In addition, we generate code that uses if or switch statements for comparison with hard-coded approaches. To benchmark the possible data structures, we use a dedicated benchmark application written in C++ that simulates the dispatching process using a single instance per data structure. For our measurements we employ the Google Benchmark library[7]. We measure dispatching time, startup time, and consider memory usage as well as caching behavior for each data structure. The startup time is the time required to build a data structure. The dispatching time is the overall time spent for identifier lookups. Startup time and memory usage solely depend on the configured identifier mappings. The dispatching time is also influenced by the monitored network traffic, as its structure determines the sequence of dispatching steps. We describe the traffic structure using a simplified, CSV-based trace format: Each line corresponds to a packet and each field to a layer, with the values representing protocol identifiers. All measurements are performed on a system equipped with a 2.6GHz Intel® Core® i7-6600U processor (2 cores, 64 KiB L1d cache, 512 KiB L2 cache) and 20 GB DDR4 memory. The benchmark application was compiled using clang version 10.0.1.

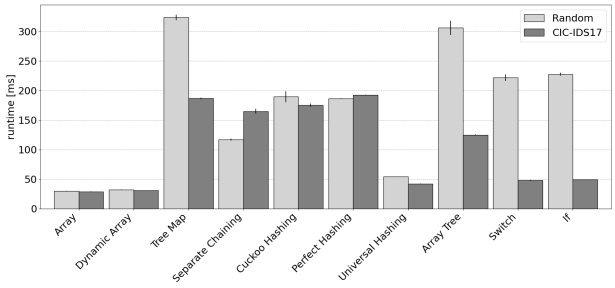
In our benchmarks we consider both, the traffic composition and the identifier mapping. For each of these dimensions we simulate two scenarios, one that aims to capture the characteristics of a real-world deployment and one that simulates a deliberately adverse situation. Aiming to capture the properties of real-world traffic, we obtain a trace from the Monday PCAP of the CIC-IDS17 data set [38]. Following the CIC-IDS17 traffic, the adverse trace consists of packets that contain three packet-level PDUs each, but whose identifiers are randomly drawn from a uniform distribution in the range of 1 to 10.000. In practice, a similar situation could occur due to incorrect parsing or being triggered intentionally by an attacker. For comparability both traces comprise a total number of 10 million PDUs to dispatch. The real-world-oriented identifier mapping covers the protocols that would be supported in a default configuration of Zeek. To account for a challenging case, we use a deliberately fragmented mapping that covers hundred identifiers found in the randomized trace as well as the identifiers of the Zeek mapping.

It is not surprising that the data structures based on more advanced hash functions come with a significant performance cost, as can be seen in Figure 5a. Given the small set of involved identifiers, the tree-based data structures perform better. The best performance is achieved by the array approaches, which may even outperform the hard-coded variants depending on their realization by the compiler. Although it is a common approach to translate larger control-flow structures into jump tables that resemble arrays, compilers may chose

[7] https://github.com/google/benchmark



(a) Concise identifier mapping based on Zeek defaults under CIC-IDS17 and randomized traffic composition.



(b) Fragmented identifier mapping (Zeek IDs plus 100 IDs from random trace) under CIC-IDS17 and randomized traffic composition.

Figure 5: Dispatching times for selected data structures. Each bar corresponds to the average of 10 runs. Error bars depict the 95% confidence intervals.

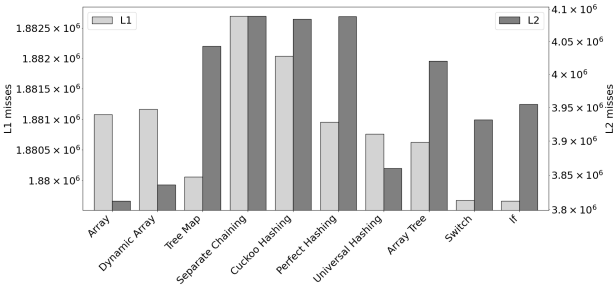


Figure 6: L1 and L2 cache misses using the trace based on the CIC-IDS17 dataset with an identifier mapping that corresponds to Zeek's default protocol support. Each bar corresponds to the average of 10 runs. Note that the differences in absolute values are small.

to realize different techniques like a binary search, with respect to the memory trade-off for sparse jump tables[8]. This explains the relatively poor performance of the hard-coded variants for the randomized trace using the fragmented mapping as depicted in Figure 5b. Likewise, the performance of the tree-based data structures foreseeably degrades for the fragmented mapping. The universal hashing approach is able to keep up with the array approaches due to the simplicity of the employed hash function.

[8] While we observed significant differences between clang and GCC, both compilers allow for modification of their standard behavior.

In terms of memory consumption, as expected, the tree-based and hash-based approaches scale well for increasing numbers of identifiers. In contrast, the static array consumes a fixed amount of memory that remains largely unused. For compact mappings, the memory consumption can be improved by using a dynamic array trimmed at front and end. Yet, the memory utilization remains poor for widely scattered mappings. However, assuming two byte identifiers and a 64 bit platform, the maximal size of an array would be 512 KiB, which is still negligible for common deployments. With respect to the partially significant memory consumption, we also consider the caching behavior. In general, the handling of identifier mappings can be assumed to be cache-friendly due to their static nature. Figure 6 shows the number of cache misses for the evaluated data structures[9]. Although the sparse arrays cause relatively high numbers of L1 data cache misses compared to other data structures, the comparatively good performance for L2 cache misses suggest that their simple structure is ultimately beneficial for caching. Even under the deteriorated conditions, the caching behavior does not compromise lookup performance of arrays.

Finally, startup time becomes relevant for the universal hashing approach. While the other data structures can be built in just a few milliseconds, finding a collision free hash function becomes disproportionately time consuming with the size of the identifier mappings increasing. All in all, the advantages of arrays outweigh their fairly high memory consumption, especially with regard to the predictability of the achieved performance.

5.2 Monitor Performance

Based on the previous results, we select the dynamic array for our implementation of the extended NSM architecture in Zeek. The dispatching performance for the dynamic array is comparable to the hard-coded approaches. Furthermore, the implementation is straightforward, which supports maintainability.

To measure the impact on the monitor's overall performance, we broke down the previously hard-coded analysis of packet-level protocols in Zeek into packet analyzers. The migration to packet analyzers is discussed in more detail in Section 6.1. For both versions, the original hard-coded one and the modularized version, we compared execution times and maximum memory usage for processing the CIC-IDS17 Monday PCAP, using Zeek's default configuration. The results as depicted in Table 2 show that the impact of the extended architecture is negligible with only minimal increases in processing time and memory usage.

[9]Given the peculiarities of the CPU architecture in terms of the relation between the hardware performance counters for different cache levels, we focus on the comparison of the data structures per cache level.

	Runtime [s]	Memory [MB]
Original Zeek	8.31	108.46
Extended Zeek	8.39	110.99
Difference [%]	0.8	2.3

Table 2: Comparison between original Zeek using hard-coded packet-level analysis and Zeek implementing the extended architecture, when processing the CIC-IDS17 Monday trace (1M packets). The numbers are the average of 10 runs.

6 Practical Applications

In this section we showcase the new possibilities offered by the modularization of packet-level analysis. With respect to the requirements as defined in Section 4.2, we demonstrate the benefits of *configurability* (R4) and illustrate the *universality* (R2) of the extended NSM architecture on the example of Zeek's previously hard-coded analysis of packet-level protocols. Furthermore, we extend the Spicy parser generator toolchain to facilitate the development of packet-level dissectors. Then, we discuss two practical applications for monitoring industrial control systems, presenting packet analyzers for GOOSE and ProfinetIO that exploit the *extensibility* (R1) of the proposed architecture focusing on *independency* of PA-Plugins (R3). Finally, we introduce the logging of unknown protocols to increase visibility and prevent unnoticed monitor evasions in light of *configurability* (R4).

6.1 Traditional Protocol Stack Migration and Spicy Parser Generator

As part of the implementation of the extended NSM architecture in Zeek, we migrated Zeek's previously hard-coded processing of packet-level protocols. In Section 5.2, we showed that modularization introduces only a negligible performance overhead compared to the hard-coded version (R5). In the following, we provide details on the realization of *configurability* (R4) and demonstrate the *universality* (R2) of the proposed architecture based on the migrated protocol stack. Additionally, we discuss the extension of the Spicy parser generator to aid the development of new packet-level dissectors.

Traditional Protocol Stack Migration Zeek's original implementation covers eleven packet-level network protocols: Ethernet, FDDI, IEEE 802.11, PPP, PPPoE, VLAN, MPLS, ARP, IPv4, and IPv6 as well as GRE. In addition, Zeek also supports a set of capture-related headers: LinuxSLL, NFLOG, BSD loopback encapsulation (NULL), and IEEE 802.11 RadioTap. Furthermore, we add a special analyzer that allows for skipping a configured amount of bytes. The skip analyzer enables NSM operators to quickly adapt to complex setups that exhibit fixed but unknown headers often introduced by proprietary protocols. Figure 7 shows the protocol

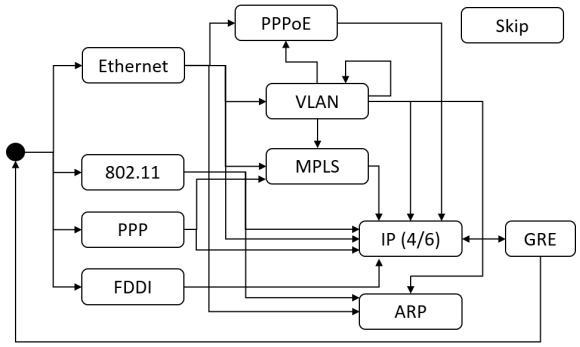


Figure 7: Protocol transitions of Zeek's traditional packet-level stack. The capture-related protocols are omitted for readability and handling of tunnels is simplified.

transitions of the traditional packet-level protocol stack. Employing the new framework for modular packet analysis, the depicted transitions are entirely configurable by the NSM operator using policy scripts, which act as Zeek's primary user interface. Listing 1 provides an excerpt from the PA-Config script for the Ethernet packet analyzer. The script defines the transition table that maps protocol identifiers, in this case EtherTypes, to the corresponding packet analyzers to be registered during initialization of Zeek. Registration is done using the `register_packet_analyzer` function that receives the parent analyzer (`ANALYZER_ETHERNET`) to register a mapping for and an identifier together with the corresponding child analyzer[10]. To customize the analysis process, operators only need to adapt the given transition tables. Overall, the *configurability* (R4) achieved by migrating Zeek's traditional protocol stack to the extended NSM architecture enables the adaption of the previously hard-coded analysis process as well as the reuse of the existing protocol analyzers in new contexts.

While we transfer the traditional stack without implementing additional parsing functionality, the *universality* (R2) of the proposed architecture allows us to account for previously unaddressed special cases. For example, Zeek's original Ethernet implementation focused on Ethernet II frames only. However, depending on the range the EtherType value falls into and the first two bytes of the payload, Ethernet frames can be differentiated into Novell raw IEEE 802.3, IEEE 802.2 LLC, IEEE 802.2 SNAP, and Ethernet II frames. By allowing the Ethernet analyzer to call configurable sub analyzers, we employ the proposed architecture to offer possibilities for future extension.

In addition to the modularization of data-link and internet layer protocol dissectors, the extended architecture also lays the foundation for the flexibilization of transport layer protocol dissectors that perform the transition from packet- to session-level. With respect to the reference architecture, there are two options to realize this transition: On the one hand, packet analyzers may implement protocol-specific session tracking by moving the session table logic into the packet analyzer itself. On the other hand, since session tracking is a common task, the monitor core may provide a generic session table component to be shared by multiple analyzers. While out of scope for this work, the Zeek project has already begun work on the latter approach.

[10]Note that Zeek 4.0 uses multiple calls to the registration function, omitting tables.

Listing 1: PA-Config excerpt that shows the implementation of the transition table for the Ethernet packet analyzer. The transition table maps EtherTypes to packet analyzers. During initialization, the defined mappings are registered.

```
global ether_types: table[count]
of PacketAnalyzer::Tag = {
    [0x8847] = PacketAnalyzer::ANALYZER_MPLS,
    [0x0800] = PacketAnalyzer::ANALYZER_IP,
    [0x86DD] = PacketAnalyzer::ANALYZER_IP,
    [0x0806] = PacketAnalyzer::ANALYZER_ARP,
    [0x8035] = PacketAnalyzer::ANALYZER_ARP,
    [0x8100] = PacketAnalyzer::ANALYZER_VLAN,
    [0x88A8] = PacketAnalyzer::ANALYZER_VLAN,
    [0x9100] = PacketAnalyzer::ANALYZER_VLAN,
    [0x8864] = PacketAnalyzer::ANALYZER_PPPOE
  } &redef;

event zeek_init() &priority=20
  {
  for ( id, child_analyzer in ether_types )
    PacketAnalyzer::register_packet_analyzer(
      PacketAnalyzer::ANALYZER_ETHERNET,
      id, child_analyzer);
  }
```

Spicy Parser Generator Though the refined NSM architecture allows for easy addition of new dissectors, implementing the parsing logic still remains a cumbersome and error-prone process [37]. As a network security monitor needs to deal with arbitrary network traffic from potentially malicious sources, the robustness of packet analyzers is a major concern. To further aid the development of packet-level dissectors, we have added support for packet analyzers to the Spicy parser generator [41]. Spicy defines a language to integrate the specification of syntax and semantics for data formats ranging from network protocols to file formats and offers tooling to generate dissectors based on the format specifications. Furthermore, the Spicy toolchain already integrates with Zeek. Listing 2 shows the specification for a packet-level analyzer that parses IEEE 802.11Q VLAN tags. The first 16 bits comprise the Tag Control Information (TCI), which is divided into the VLAN Identifier (VID), the Drop Eligible Indicator (DEI), and a Priority Code Point (PCP). Then the actual EtherType is parsed and used to forward the remaining data to the next analyzer, once parsing the tag is done. Spicy being able to translate high-level specifications into a

Listing 2: Functional Spicy example that specifies a Zeek packet analyzer for 802.1Q VLAN tags.

```
module VLAN;
import zeek;
public type Packet = unit {
    tci: bitfield(16) {
        vid: 0..11;
        dei: 12;
        pcp: 13..15;
    };
    ether_type: uint16;

    on %done {
        zeek::forward_packet(self.ether_type);
    }
};
```

pluggable dissector completely eliminates the need to write C++ code and thus vastly simplifies the development of packet analyzers. Especially with respect to the binary nature of packet-level protocols, this approach also promises to yield more robust protocol parsers, which is crucial for the overall security of the monitor.

6.2 Monitoring in Industrial Control Systems

In the context of Industrial Control Systems (ICS), the integration of so-called Operational Technology (OT) and traditional Information Technology (IT) increases. Connecting ICS to larger networks or even the internet introduces new attack vectors that represent substantial threats, e.g., when exposing critical infrastructures [30, 17]. State-of-the-art NSMs typically lack the ability to provide visibility into domain-specific ICS protocols. In the following, we show how the proposed modular NSM architecture can be used to extend a network security monitor to make information extracted from ICS communication available for security operations by exploiting the *independency* (R3) of PA-Plugins. To this end, we extend Zeek with packet-level analyzers for GOOSE, used in the energy sector, and ProfinetIO, employed in production. We briefly introduce each protocol, present attacks discussed in recent research, and demonstrate the value of integrating modular packet-level dissectors by leveraging them to implement corresponding attack detection logic in Zeek's domain-specific scripting language.

GOOSE Protocol The Generic Object-Oriented Substation Events protocol (GOOSE) is used for the communication between Intelligent Electronic Devices (IEDs) of electrical substations. The protocol is standardized as part of IEC 61850. To meet real-time requirements, GOOSE builds directly on top of Ethernet and applies a multicast communication scheme. Because GOOSE

Listing 3: Configuration for the integration of GOOSE.

```
redef PacketAnalyzer::ETHERNET::ether_types +=
{
  [0x88b8] = PacketAnalyzer::ANALYZER_GOOSE,
  [0x88b9] = PacketAnalyzer::ANALYZER_GOOSE
};
redef PacketAnalyzer::VLAN::protocols +=
{
  [0x88b8] = PacketAnalyzer::ANALYZER_GOOSE,
  [0x88b9] = PacketAnalyzer::ANALYZER_GOOSE
};
```

lacks authentication mechanisms, the protocol is susceptible to machine-in-the-middle attacks. Attackers with access to the substation network may inject false data to interfere with the operation of the power grid.

The need for visibility into GOOSE communication is well recognized. Kabir-Querrec already extended Zeek to parse GOOSE messages [25]. Recently, another patch has been released by the ResiGate project of the Advanced Digital Sciences Center (ADSC) [8]. However, due to the far-reaching changes these extensions require to the monitor core, the code was not officially integrated into Zeek. The proposed extended architecture for modular packet-level analysis allows us to decouple the dissection of the GOOSE protocol and the monitor core, by moving the existing code into a PA-Plugin[11]. Considering that GOOSE packets are identified by two EtherTypes (`0x88b8` and `0x88b9`) and can be found either directly in Ethernet frames or using VLANs, again, the *configurability* (R4) of the proposed architecture proves to be advantageous. Listing 3 shows the corresponding configuration for GOOSE. Note that the existing transition tables are extended using a Zeek language construct (`redef`) that allows for the addition of new entries to existing tables.

Due to the limited practical feasibility of mechanisms to secure GOOSE communication, a lot of recent research particularly addresses attack detection [23, 25, 4]. For example, Bohara et al. discuss a so-called poisoning attack [4]: During normal operation, GOOSE endpoints regularly send messages announcing their current state. Each message also contains two counters, state number (st) and sequence number (sq). For each repeated message sq is increased. In case of an event that changes the encoded state, st is increased to signal the state change. After a state change, messages are sent in a higher frequency before exponentially slowing down to the original sending interval. Based on the counter values, endpoints discard already processed messages to reduce load. In a simple attack, valid messages can be "overwritten" by an attacker sending out messages with high counter values. Monitoring the progression of the counter values, this type of attack can be easily detected.

[11] https://github.com/kit-dsn/zeek-goose-analyzer

Listing 4: Exemplary Zeek script to detect $st' > st + 1$

```
global stNums: table[string] of count;

event goose_message(info: GOOSE::PacketInfo,
    pdu: GOOSE::PDU) {
  local ds: string = pdu$datSet;
  # Initialization
  if ( ds !in stNums ) {
    stNums[ds] = pdu$stNum;
    return;
    }
  # Check for increments > 1
  if ( pdu$stNum > stNums[ds] + 1 )
    print fmt("State number jump to %d",
      pdu$stNum);
  # Update counter
  stNums[ds] = pdu$stNum;
  }
```

Given the *independency* (R3) of PA-plugins, the modular GOOSE dissector is able to generate an event for each GOOSE message. The event receives the message content including the counter values as an argument so that policies can be defined with respect to these values. Listing 4 shows a simplified example of an event handler that detects jumps of the state number (*st*) larger than one. We track the counters per GOOSE data set in the `stNums` table. In case the data set has not been seen before, the counter values are initialized. Otherwise, we check for increments larger one and print a message if we detect a jump. Finally, the tracked counters are updated. For brevity, we omit the handling of counter rollovers. Furthermore, attackers may adapt to the currently valid counter numbers. Bohara et al. present an advanced detection logic that detects inadequately overlapping resends, which indicate the insertion of messages using valid counters by a third party [4]. In this regard, the extended architecture for packet-level analysis promotes the transfer of current research into practice by establishing a clear separation of concerns. On the one hand, dissecting the GOOSE protocol is decoupled from the monitor core and moved into an easy to deploy PA-Plugin to be maintained separately. On the other hand, the demand-driven integration of the dissector in the overall monitoring system allows security experts to exploit the synergies of a unified interface, as provided by the domain-specific Zeek scripting language, to implement and share new detection approaches.

ProfinetIO Protocol In a typical production scenario a process is implemented using field devices such as sensors and actuators that are controlled by Programmable Logic Controllers (PLCs). ProfinetIO handles the communication between PLCs and field devices as well as other components such as programming stations or human-machine interfaces. With more than 32 million deployed nodes [43], ProfinetIO is one of the most widely

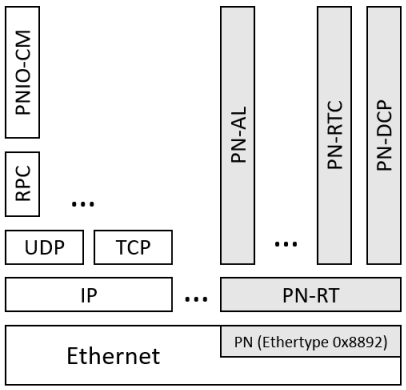


Figure 8: Profinet protocol suite (c.f. [29]). IP-based protocols are supported while the Ethernet-based stack is unsupported by traditional NSMs.

used Ethernet-based fieldbus protocols and was standardized as part of IEC 61158. The Profinet protocol suite (c.f. Figure 8) can be divided into several protocols and sub-protocols. In particular, the following three protocols need to be considered: The Discovery and Configuration Protocol (DCP), the Context Management protocol (CM), and the Real Time Cyclic protocol (RTC). During the start up phase of a Profinet system, PLCs use the Ethernet-based DCP to discover field devices by name and provide an IP configuration for these devices. Subsequently, the UDP-based CM protocol can be used to configure process-related communication relations between devices and controllers. Once the setup is completed, the actual process control communication is handled by the RTC protocol using one of three priority classes according to the previously configured relations. While existing NSMs can be readily extended to monitor the UDP-based CM communication[12], there is no visibility into the larger part of the stack that is Ethernet-based.

Pfrang and Meier investigate replay-attacks against the Ethernet-based ProfinetIO [32]. To successfully replay traffic, they disrupt the existing communication relationship between a field device and its controlling PLC by conducting a reconfiguration attack. In a reconfiguration attack DCP is used to rename a victim device, which terminates existing connections. Afterwards, the attacker can establish new communication relations to execute a subsequent replay attack. By monitoring the Ethernet-based DCP, these attacks can be easily detected as well.

Leveraging the extended NSM architecture for modular packet-level analysis, we implement an analyzer for ProfinetIO focusing on DCP. DCP uses the common PN-RT header that is transferred over Ethernet and identified by EtherType `0x8892`. Again, we exploit the *independency* (R3) of PA-plugins to enable the definition of DCP-specific policies by generating events that reflect the request-response-oriented com-

[12]For example, Amazon's *Zeek Plugin PROFINET* [2]

munication flow. For our implementation we employ the previously introduced Spicy parser generator, which allows for the specification of Zeek events to be triggered based on the progress of parsing. To detect successful renaming attacks, we track the configured values of the observed devices to distinguish between benign rewrites and malicious changes of device names. This is done by correlating renaming requests, responses that confirm renaming, and subsequent searches for the old names. We provide both, the ProfinetIO packet analyzer and the detection script, in a separate Zeek plugin[13].

With respect to the complexity of the Profinet protocol stack that is characterized by the interactions and dependencies of the different protocols, being able to gather and correlate information from the various subsystems offers new possibilities for comprehensive, system-scope analyses. Yet simple, the example attack detectors for GOOSE and ProfinetIO clearly demonstrate the value of modular packet analyzers. The separation of policy and mechanism (c.f. [31]) using a domain-specific language for monitoring facilitates the access of security experts to new application areas and promotes the exchange and transfer of existing knowledge. In addition to detecting known attack patterns, NSM operators might choose to apply environment-specific policies that flag deviations from expected behavior. Furthermore, machine learning approaches could be applied to implement anomaly detection. In this context, we would like to emphasize again that we see our work as complementary to the use of machine learning methods, since the quality of these methods significantly depends on the selection and preprocessing of input data [3].

6.3 Unknown Protocols

Visibility is paramount in network security monitoring, but, as explained in Section 3, it is neither feasible nor desirable to support monitoring of all protocols. Because protocol parsing itself is complex, unnecessary dissectors increase the monitor's attack surface in terms of implementation errors. Furthermore, attackers might be able to degrade the monitor's performance by confronting an NSM with network traffic that is specially crafted to trigger expensive processing using otherwise irrelevant protocols. Once protocol identification becomes ambiguous, the simultaneous operation of multiple protocol dissectors might even be exploited to evade the monitor [20]. However, making use of *configurability* (R4) by just excluding protocols would immediately introduce a new attack vector, if attacks using these protocols go completely unnoticed.

To prevent excluded protocols from being exploited to bypass the monitor, we log the use of unknown protocols. In the *unknown_protocols.log*, the time, the analyzer that encountered an unknown protocol, the corresponding protocol identifier, and a configurable amount of bytes from the unknown PDU are logged. If a protocol unknown to the NSM is regularly used in the monitored network, logging each unknown packet would quickly overwhelm the monitor. Thus, we also introduced a throttling mechanism. For each protocol analyzer, the packets per unknown protocol identifier are counted. Once the threshold is reached, the reporting of packets using this unknown protocol identifier is rate-limited to a given sampling rate. The rate-limiting expires after a specified duration. Threshold, sampling rate, and rate-limiting duration are configurable as well.

We expect the logging of unknown protocols to significantly improve the deployment of an NSM in new environments. By evaluating the unknown protocols log, NSM operators can quickly determine if additional protocols need to be supported or spot misconfigurations. In a parallel effort during our work, ESnet[14] already proved the value of unknown protocol detection in practical operations.

7 Conclusion

With the increasing proliferation of information technology in new areas such as energy, mobility, and production, NSMs need to cope with the corresponding diversification of the TCP/IP-oriented protocol stack. While existing approaches for modularization of protocol dissection at application layer provide a high degree of flexibility, they do not extend to packet-level analysis, leaving established NSMs incapable of dynamically integrating non-IP protocol dissectors. Given that processing at these low levels is highly performance critical, the question arises whether it is practically feasible to replace the hard-coded path with a dynamic plugin scheme.

In this paper, we present a refined software architecture for NSMs that enables the flexible integration of lower-layer protocol dissectors. We investigate the challenges in the context of packet-level dispatching and derive detailed requirements for the implementation of the proposed, extended architecture. Instead of the prevalent notion of well-ordered protocol stacks, in practice the interrelations between protocols form more complex protocol transition graphs. With dispatching performance being of utmost importance, we find that hash-based data structures introduce a significant overhead, whereas array-based approaches can keep up with the hard-coded processing at packet-level. Based on this result, we implement the extended NSM architecture in the open-source NSM Zeek, migrate Zeek's previously hard-coded stack of packet-level protocols to the new interface, and verify that the performance impact is indeed negligible. We demonstrate the benefits of the proposed architecture by implementing basic attack detection

[13] https://github.com/kit-dsn/zeek-profinet-analyzer

[14] https://www.es.net/

techniques as proposed in recent research on the security of industrial control systems in the fields of energy and production, employing the new interface to realize modular dissectors for GOOSE and ProfinetIO, two popular industrial communication protocols. To further assist the practical application of modular packet-level protocol dissectors, we integrate with the Spicy parser generator, which allows for creating new dissectors in a declarative fashion, and present a feature to increase visibility when a monitor is confronted with unknown protocols.

With our work, we hope to support the scientific community as well as practitioners. In light of the rapid evolution and increasing adoption of information technology into ever new environments, we seek to extend visibility for security operations, facilitate the adoption of existing technologies and, in particular, promote the transfer of application-domain-driven research into practice. Thus, all artifacts are made publicly available including the implementation of the extended NSM architecture for modular packet-level analysis being part of Zeek's 4.0 release.

Acknowledgments

Hannes Hartenstein and Jan Grashöfer acknowledge the funding of the Helmholtz Association (HGF) through the Competence Center for Applied Security Technology (KASTEL).

References

[1] Johanna Amann, Seth Hall, and Robin Sommer. 2014. Count me in: viable distributed summary statistics for securing high-speed networks. In *Research in Attacks, Intrusions and Defenses*. Volume 8688. Springer International Publishing. DOI: 10.1007/978-3-319-11379-1_16.

[2] Amazon.com, Inc. 2021. Zeek plugin PROFINET. Retrieved 03/31/2021 from https://github.com/amzn/zeek-plugin-profinet.

[3] Blake Anderson and David McGrew. 2017. Machine learning for encrypted malware traffic classification: accounting for noisy labels and non-stationarity. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, (August 13, 2017). DOI: 10.1145/3097983.3098163.

[4] Atul Bohara, Jordi Ros-Giralt, Ghada Elbez, Alfonso Valdes, Klara Nahrstedt, and William H. Sanders. 2020. ED4gap: efficient detection for GOOSE-based poisoning attacks on IEC 61850 substations. In *2020 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*. IEEE, (November 11, 2020). DOI: 10.1109/SmartGridComm47815.2020.9303015.

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, (July 28, 2014). DOI: 10.1145/2656877.2656890.

[6] Tomash Brechko. 2013. Cuckoo hash. Retrieved 03/22/2021 from https://github.com/kroki/Cuckoo-hash.

[7] Valentina Casola, Alessandra De Benedictis, Antonio Riccio, Diego Rivera, Wissam Mallouli, and Edgardo Montes de Oca. 2019. A security monitoring system for internet of things. *Internet of Things*. DOI: https://doi.org/10.1016/j.iot.2019.100080.

[8] Binbin Chen, Heng Chuan Tan, and Vyshnavi M. 2020. Goose protocol parser for zeek IDS. Advanced Digital Sciences Center (ADSC). Retrieved 03/31/2021 from https://github.com/smartgridadsc/Goose-protocol-parser-for-Zeek-IDS.

[9] Justyna Chromik, Anne Remke, Boudewijn R. Haverkort, and Gerard Geist. 2019. A parser for deep packet inspection of IEC-104: a practical solution for industrial applications. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Industry Track*. (June 2019). DOI: 10.1109/DSN-Industry.2019.00008.

[10] Rafał Cupek, Markus Bregulla, and Łukasz Huczała. 2009. PROFINET i/o network analyzer. In *Computer Networks*. Series Title: Communications in Computer and Information Science. Springer Berlin Heidelberg. DOI: 10.1007/978-3-642-02671-3_29.

[11] Cybersecurity and Infrastructure Security Agency (CISA) and Idaho National Laboratory (INL). 2021. ICSNPP-BSAP-serial. Retrieved 03/31/2021 from https://github.com/cisagov/icsnpp-bsap-serial.

[12] Lorenzo De Carli, Robin Sommer, and Somesh Jha. 2014. Beyond pattern matching: a concurrency model for stateful deep packet inspection. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*. DOI: 10.1145/2660267.2660361.

[13] S. Deering and R. Hinden. 2017. Internet Protocol, Version 6 (IPv6) Specification. RFC8200. RFC Editor, (July 2017). DOI: 10.17487/RFC8200.

[14] Holger Dreger, Anja Feldmann, Michael Mai, Vern Paxson, and Robin Sommer. 2006. Dynamic application-layer protocol analysis for network intrusion detection. In *15th USENIX security symposium*. USENIX Association, 257–272.

[15] D. Eastlake and J. Abley. 2013. IANA Considerations and IETF Protocol and Documentation Usage for IEEE 802 Parameters. RFC7042. RFC Editor, (October 2013). DOI: 10.17487/rfc7042.

[16] Donald Eastlake and Juan Carlos Zuniga. 2021. IEEE 802 numbers. (March 1, 2021). Retrieved 03/31/2021 from https://www.iana.org/assignments/ieee-802-numbers/ieee-802-numbers.xhtml.

[17] Ghada Elbez, Hubert B. Keller, and Veit Hagenmeyer. 2018. A new classification of attacks against the cyber-physical security of smart grids. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*. (August 27, 2018). DOI: 10.1145/3230833.3234689.

[18] Edward A. Fox, Lenwood S. Heath, Qi Fan Chen, and Amjad M. Daoud. 1992. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, (January 2, 1992). DOI: 10.1145/129617.129623.

[19] Glen Gibb, George Varghese, Mark Horowitz, and Nick McKeown. 2013. Design principles for packet parsers. In *2013 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. (October 2013). DOI: 10.1109/ANCS.2013.6665172.

[20] Jan Grashofer, Christian Titze, and Hannes Hartenstein. 2020. Attacks on dynamic protocol detection of open source network security monitoring tools. In *2020 IEEE Conference on Communications and Network Security (CNS)*. (June 2020). DOI: 10.1109/CNS48642.2020.9162332.

[21] Steve Hanov. 2011. Throw away the keys: easy, minimal perfect hashing. Retrieved 03/22/2021 from http://stevehanov.ca/blog/?id=119.

[22] Zachary Hill, John Hale, Mauricio Papa, and Peter Hawrylak. 2019. Using bro with a simulation model to detect cyber-physical attacks in a nuclear reactor. In *2019 2nd International Conference on Data Intelligence and Security (ICDIS)*. (June 2019). DOI: 10.1109/ICDIS.2019.00011.

[23] Juan Hoyos, Mark Dehus, and Timthy X Brown. 2012. Exploiting the GOOSE protocol: a practical attack on cyber-infrastructure. In *2012 IEEE Globecom Workshops*. (December 2012). DOI: 10.1109/GLOCOMW.2012.6477809.

[24] Ray Hunt and Sherali Zeadally. 2012. Network forensics: an analysis of techniques, tools, and trends. *Computer*, (December 2012). DOI: 10.1109/MC.2012.252. Retrieved 12/07/2020 from.

[25] Maëlle Kabir-Querrec. 2017. *Cyber security of smart-grid control systems: intrusion detection in IEC 61850 communication networks*. PhD thesis.

[26] Jack Koziol. 2003. *Intrusion detection with Snort*. Sams Publishing. ISBN: 978-1-57870-281-7.

[27] Hongda Li, Hongxin Hu, Guofei Gu, Gail-Joon Ahn, and Fuqiang Zhang. 2018. vNIDS: towards elastic security with safe and efficient virtualization of network intrusion detection systems. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. (January 15, 2018). DOI: 10.1145/3243734.3243862.

[28] Hui Lin, Adam Slagell, Catello Di Martino, Zbigniew Kalbarczyk, and Ravishankar Iyer. 2013. Adapting bro into SCADA: building a specification-based intrusion detection system for the DNP3 protocol. In *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop on - CSIIRW '13*. ACM Press. DOI: 10.1145/2459976.2459982.

[29] Alexandr Osadcii. 2017. Design and implementation of automatic tests for siemens PROFINET IO development kit.

[30] Andreas Paul, Franka Schuster, and Hartmut König. 2013. Towards the protection of industrial control systems – conclusions of a vulnerability analysis of profinet IO. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. DOI: 10.1007/978-3-642-39235-1_10.

[31] Vern Paxson. 1999. Bro: a system for detecting network intruders in real-time. *Computer Networks*, 31. http://www.icir.org/vern/papers/bro-CN99.pdf.

[32] Steffen Pfrang and David Meier. 2018. Detecting and preventing replay attacks in industrial automation networks operated with profinet IO. *Journal of Computer Virology and Hacking Techniques*, (November 2018). DOI: 10.1007/s11416-018-0315-0.

[33] D. Plummer. 1982. Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware. RFC0826. RFC Editor, (November 1982). DOI: 10.17487/rfc0826.

[34] Snort Project. 2021. Snort3 devnotes. Retrieved 03/31/2021 from https://github.com/snort3/snort3/blob/master/src/codecs/dev_notes.txt.

[35] Thomas H Ptacek and Timothy N Newsham. 1998. Insertion, evasion, and denial of service: Eluding network intrusion detection. Secure Networks Inc.

[36] E. Rosen, D. Tappan, G. Fedorkow, Y. Rekhter, D. Farinacci, T. Li, and A. Conta. 2001. MPLS Label Stack Encoding. RFC3032. RFC Editor, (January 2001). DOI: 10.17487/rfc3032.

[37] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Anna Shubina. 2011. The halting problems of network stack insecurity. *;login:* 36.

[38] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. 2018. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *Proceedings of the 4th International Conference on Information Systems Security and Privacy*. DOI: 10.5220/0006639801080116.

[39] W. Simpson. 1994. The Point-to-Point Protocol. RFC1661. RFC Editor, (July 1994). DOI: 10.17487/rfc1661.

[40] Robin Sommer. Following the packets: a walk through bro's processing pipeline. (2017). Retrieved 03/31/2021 from https://old.zeek.org/brocon2017/slides/bro_internals.pdf.

[41] Robin Sommer, Johanna Amann, and Seth Hall. 2016. Spicy: a unified deep packet inspection framework for safely dissecting all your data. In *Proceedings of the 32nd Annual Conference on Computer Security Applications - ACSAC '16*. ACM Press. DOI: 10.1145/2991079.2991100.

[42] Robert Udd, Mikael Asplund, Simin Nadjm-Tehrani, Mehrdad Kazemtabrizi, and Mathias Ekstedt. 2016. Exploiting bro for intrusion detection in a SCADA system. In *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security - CPSS '16*. DOI: 10.1145/2899015.2899028.

[43] Barbara Weber. 2020. PROFINET and IO-link on the rise. PROFIBUS and PROFINET International (PI). (May 6, 2020). Retrieved 03/31/2021 from https://www.profibus.com/newsroom/press-news/profinet-and-io-link-on-the-rise.

[44] Philipp Woelfel. 1999. Efficient strongly universal and optimally universal hashing. In *Mathematical Foundations of Computer Science 1999*. DOI: 10.1007/3-540-48340-3_24.

[45] Kevin Wong, Craig Dillabaugh, Nabil Seddigh, and Biswajit Nandy. 2017. Enhancing suricata intrusion detection system for cyber security in SCADA networks. In *2017 IEEE 30th Canadian Conference on Electrical and Computer Engineering (CCECE)*. (April 2017). DOI: 10.1109/CCECE.2017.7946818.

[46] Shitong Zhu, Shasha Li, Zhongjie Wang, Xun Chen, Zhiyun Qian, Srikanth V. Krishnamurthy, Kevin S. Chan, and Ananthram Swami. 2020. You do (not) belong here: detecting DPI evasion attacks with context learning. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. ACM, (November 23, 2020). DOI: 10.1145/3386367.3431311.

[47] Zihao Feng, Sujuan Qin, Xuesong Huo, Pei Pei, Ye Liang, and Liming Wang. 2016. Snort improvement on profinet RT for industrial control system intrusion detection. In *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. (October 2016). DOI: 10.1109/CompComm.2016.7924843.