ED4GAP: Efficient Detection for GOOSE-Based Poisoning Attacks on IEC 61850 Substations

Atul Bohara*, Jordi Ros-Giralt†, Ghada Elbez‡[1], Alfonso Valdes*,
Klara Nahrstedt*, and William H. Sanders*
*Information Trust Institute (University of Illinois at Urbana-Champaign),
†Reservoir Labs, and ‡Institute of Automation and Applied Informatics (Karlsruhe Institute of Technology)

***Abstract*—Devices in IEC 61850 substations use the generic object-oriented substation events (GOOSE) protocol to exchange protection-related events. Because of its lack of authentication and encryption, GOOSE is vulnerable to man-in-the-middle attacks. An adversary with access to the substation network can inject carefully crafted messages to impact the grid's availability. One of the most common such attacks, GOOSE-based poisoning, modifies the `StNum` and `SqNum` fields in the protocol data unit to take over GOOSE publications. We present ED4GAP, a network-level system for efficient detection of the poisoning attacks. We define a finite state machine model for network communication concerning the attacks. Guided by the model, ED4GAP analyzes network traffic out-of-band and detects attacks in real-time. We implement a prototype of the system and evaluate its detection accuracy. We provide a systematic approach to assessing bottlenecks, improving performance, and demonstrating that ED4GAP has low overhead and meets GOOSE's timing constraints.**

I. Introduction

The primary purpose of the power grid protection system is to minimize the impact of electrical faults. Substations in the modern power grid achieve this objective by using real-time communication between intelligent electronic devices (IEDs). In IEC 61850-compliant systems, the protection-related data exchange relies on the generic object-oriented substation events (GOOSE) protocol [1].

The security of GOOSE communication is a crucial aspect of the grid's reliability. Accordingly, the IEC 62351 standard [2] was developed to fill the security gaps in GOOSE, among other protocols. However, high-speed communication, coupled with the resource-constrained substation network, has resulted in limited application of the proposed security measures [3]. In fact, IEC 62351 does not recommend the use of encryption for GOOSE. Thus, not all of the guidelines, such as recommendations that message authentications use digital signatures and that certificates be supported in configuration files, are applicable. As a result, the substation security still relies significantly on *best practices* [4] and the implementations and configurations from individual vendors [5].

Nevertheless, researchers have explored some encryption-based measures for hardening the integrity and authenticity of GOOSE messages (e.g., [6]–[8]). All of these approaches fail to meet the most stringent GOOSE transfer time unless a significant overhaul is made in IEDs' hardware [9], [10]. Another set of methods (e.g., [9]–[12]) that can satisfy the timing constraints uses pre-shared symmetric keys, requiring the extension of key management for GOOSE or the installation of additional hardware boxes in front of each IED. Because of those obstacles, what we have today—an unencrypted channel between substation devices—makes GOOSE susceptible to man-in-the-middle (MITM) attacks. The absence of flow control and acknowledgments further increases the risk.

We primarily focus on GOOSE poisoning, a common class of MITM attacks affecting system availability [4], [5], [13], [14]. Such attacks work by injecting valid GOOSE messages that can cause IEDs to stop processing legitimate traffic and take malicious control actions.

In this paper, we introduce *ED4GAP*, a network-level system for *E*fficient *D*etection *For G*OOSE-b*A*sed *P*oisoning attacks. ED4GAP monitors the substation's local area network through port mirroring on switches. It then analyzes the GOOSE messages concurrently to the data transmission to identify indicators of poisoning attacks. The identification depends on the protocol specifications, as well as the attack's needs to inject extra packets on the network. The solution uses a state machine model to find such syntactically correct but malicious packets. The model enumerates all possible transitions, both valid and invalid, from the current state of GOOSE communication. Any transitions to the invalid states are easily detected. However, a sophisticated attacker can achieve malicious goals even if limited to only benign transitions. Our approach overcomes that challenge by introducing a *staging state*. The staging state temporarily stores the information about a set of packets until a classification of "attack" or "normal" is confirmed. Finally, the system generates alerts concerning the detected attacks to enable further action.

We implement a prototype of ED4GAP using the Zeek network security monitor [15]. With Zeek, we use a protocol analyzer [16] to decode GOOSE traffic, implement the attack detection method as a policy script, and extend the logging mechanism to produce GOOSE-related logs and attack alerts. We further devise a systematic approach to identify bottlenecks and speed-up the traffic processing throughput by using the exclusive pinning of CPU cores.

In our experiments, we generated traffic by replaying a synthesized dataset [17] representing an IEC 61850 substation.

[1] Ghada Elbez was a visiting scholar at the Information Trust Institute of the University of Illinois at Urbana-Champaign when this work was done.

978-1-7281-6127-3/20/$31.00 ©2020 IEEE

ED4GAP accurately detected all the attacks and generated corresponding alerts. Afterward, we evaluated the real-time performance of ED4GAP to ensure that it could support GOOSE timing constraints. We replayed the traffic by increasing the transmission speed to achieve a total of more than 200 experimental settings. When employed as a virtual machine on a host with an Intel Core i7 quad-core processor running at 2.6 GHz with 8 GB of RAM, ED4GAP could achieve a throughput of up to 21.98 megabits per second (Mbps) and average per-packet response time of 0.03 milliseconds (ms). For the current dataset, that throughput translated into 17,535 packets per second. Our results demonstrated that ED4GAP could analyze traffic and detect attacks while satisfying the real-time requirements of GOOSE.

The contributions of our paper include:

- a *specification-based method* for detecting GOOSE poisoning attacks even when the attacker can replicate normal behavior,
- an *implementation of ED4GAP*—a system for out-of-band concurrent traffic analysis and attack detection, and
- a *systematic approach* to finding bottlenecks and optimizing performance by varying the number and using the exclusive pinning of CPU cores.

II. Preliminaries

The GOOSE protocol is responsible for exchanging commands, alarms, and indications in IEC 61850 substations [1]. GOOSE groups such data in a construct called a *dataset* and aims to achieve a strict transfer time, which is 3 ms for the most critical events, e.g., trips and blockings. To meet the timing requirements, GOOSE sits directly above the Ethernet layer and follows a multicast communication model. In this model, a *publisher* sends data to a multicast Ethernet address to which multiple *subscribers* can register to receive the data.

A GOOSE message, encapsulated within an Ethernet frame, has a variable length of fewer than 1500 bytes, denoted by the `Length` field. The variability in length comes from GOOSE's application protocol data unit (APDU), a twelve-field sequence. Some of the fields relevant to our discussion include (1) `DatSet`, an identifier for the set of values transferred, i.e., the dataset; (2) `T`, the timestamp at which the state last changed; and (3) `TimeAllowedtoLive` (`TAL`), an indicator for a subscriber of how long to wait for the next packet.

To achieve communication reliability at high speed, GOOSE makes use of an enhanced retransmission mechanism instead of the usual acknowledgments. In the GOOSE scheme, represented in Fig. 1, a message is published immediately after a new event. Afterward, the publisher retransmits the same state as long as it persists. The retransmissions happen with an exponentially increasing time separation between messages, e.g., T_1, T_2, T_3, and so on. Once the interval has reached a steady-state value of T_0, it stops changing. The values of T_1, T_0, and the exponential increment are set at the initial configuration of the devices.

The GOOSE APDU also includes two 32-bit counters, namely a state number (`StNum`) and a sequence number (`SqNum`), to support the retransmission mechanism. Every new event in the substation causes a change in one or more values in the corresponding `DatSet`. The source device reacts to that change by publishing a GOOSE message with an incremented `StNum`. In the absence of any state changes, the publisher transmits identical messages, and only `SqNum` is incremented. While publishers set the values of `StNum`, `SqNum`, `T`, and other APDU fields, GOOSE subscribers use them to admit only new messages. More specifically, a subscriber processes messages that have a higher `StNum` or higher `SqNum` (with `StNum` unchanged) than the last message had.

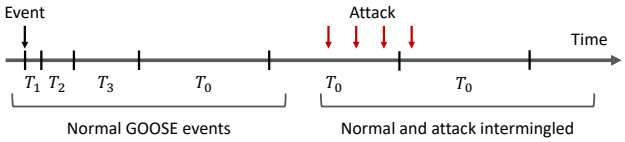


Fig. 1: Enhanced retransmission mechanism of GOOSE; T_i is interval between two successive messages.

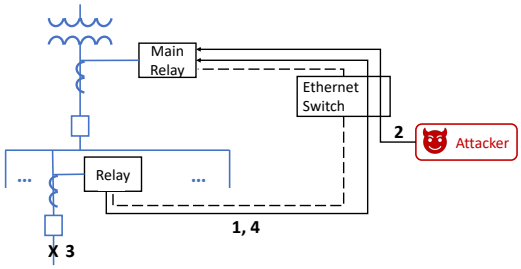


Fig. 2: Example of a GOOSE poisoning attack on the relay blocking response; numbers indicate the message sequence.

Threat Model. We consider an attacker who can monitor, spoof, and publish GOOSE messages within a substation. In particular, we assume that the adversary (1) knows subscribers' processing logic and the current status of GOOSE communication, (2) injects packets at the desired time-points, and (3) can choose values of `StNum` and `SqNum` (among other fields) such that the malicious traffic gets admitted. However, we trust the IEDs and assume that they behave as expected.

This threat model is relevant because of the vulnerabilities discussed earlier, in particular the lack of encryption and authentication. The former allows the attacker to observe the communication, whereas the latter causes the spoofed frames to be processed. Such attacks can cause a denial-of-service or the manipulation of devices. They can lead to severe consequences, including loss of electricity, damage to equipment, and injury to humans [13], [14], [16]. Therefore, these attacks are also known as *GOOSE poisoning*.

As a concrete example, we consider relays assigned to protection zones, in which correct operation is for the relay nearest a fault to trip and signal upstream relays not to trip via a blocking signal [18], sent as a GOOSE message, as shown in Fig. 2. An attacker can analyze the properties of that messaging, and after a normal operation update (event 1), can send a high-`StNum` frame (event 2). After the malicious message

has been processed, the upstream relays would discard any message that has a lower StNum, including reverse blocking (event 4) in response to a fault (event 3). As a result, the main relay might sense the fault at a later point and trip the attached breaker. Hence, the attack can cause an outage over a more extensive part of the system, i.e., trip the main circuit breaker instead of a single downstream breaker.

III. The ED4GAP System

To detect the indicators of GOOSE poisoning within a substation, first, the ED4GAP system monitors the traffic on the station bus. It can do so using port mirroring on Ethernet switches. Next, it uses the Zeek platform to decode the traffic and extract relevant features. Subsequently, it performs a specification-based security analysis to detect GOOSE poisoning attacks. Finally, it generates alerts corresponding to the detected attacks, which it can send to the substation human-machine interface or the control center for further action. In what follows, first, we discuss our design choices. We then present ED4GAP's specification-based analysis and GOOSE poisoning detection approach.

ED4GAP is a passive detection device that does not interfere with ongoing communication. Instead, it analyzes traffic data out-of-band in real-time and generates alerts about attacks. Despite that placement, in order to support high-speed GOOSE communication, it needs to ensure fast analysis. In that context, we had the following design considerations. First, the packet-analysis response time should be minimal for both benign and malicious cases. Second, the system should operate with a low overhead even with the high-speed traffic that is common in substations. Finally, the system should be easy to plug into existing environments with minimal changes.

Specification-Based Analysis. GOOSE specifications [1] define the manner in which StNum and SqNum are used when the substation devices exchange protection-related information. To devise a method for the detection of poisoning, we started by developing a state machine model for those specifications. This model is at the level of a GOOSE DatSet because the DatSet uniquely identifies the entity responsible for every transmission.

As shown in Fig. 3, the model enumerates all the states (starting at the values (a, b)) that the (StNum, SqNum) pair can take and that the subscriber would process. States "new state" and "retransmission" refer to an incremented StNum or SqNum, respectively, which are the most common situations in every GOOSE exchange. States "st rollover" and "sq rollover" correspond to the scenarios in which one of the counters rolls over and resets to its minimum value. Finally, the two invalid states correspond to an increment of greater than one in either of the counters; such increments are allowed and processed by subscribers.

Packets having lower-than-expected values for StNum or SqNum, i.e., $st < a$ or $sq < b$, are discarded by the subscribers. Similarly, devices typically can check and ignore syntactically incorrect frames, e.g., one with an invalid timestamp (T), Length-size mismatch, or invalid MAC address.

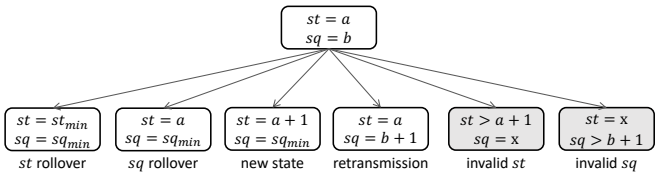


Fig. 3: DatSet-level state machine, as per GOOSE specifications. Variables st and sq represent StNum and SqNum, respectively; st_{min} and sq_{min} are the minimum values that the counters can take; and x stands for any permissible value.

Therefore, we do not include those states in the model. Finally, although GOOSE poisoning can manipulate data values in addition to StNum and SqNum, we do not consider that explicitly during the detection. However, so long as the attack involves the injection of at least one GOOSE message, our approach will detect it, as we describe next.

Detecting GOOSE Poisoning. The basic premise of our approach is that even though the attacker can inject harmful GOOSE messages into the network, the benign publishers will continue transmitting their respective states. Therefore, we can detect the attack by logically following the benign message sequence and identifying any anomalies concerning the protocol specification.

In the context of the state model (Fig. 3), it is relatively simple to find attacks that rely on the two invalid states. A detector can do so by looking for unusually high values of StNum or SqNum in an otherwise consistent sequence. However, an intruder can attack while staying within the four valid states, i.e., can inject packets at approximately the same times when benign packets are being transmitted and can contain expected or valid values for StNum and SqNum. Moreover, the attack may consist of more than one message closely following the GOOSE retransmissions. Our detector must therefore consider all those scenarios.

The detector's approach is to maintain a memory or state consisting of metadata extracted from GOOSE messages. The metadata necessary for the detection include DatSet, TAL, StNum, and SqNum, although in our implementation we also extract a message's timestamp and the source and destination MAC addresses to obtain some context that is useful for alerts. In particular, we define two states, *normal* and *staging*. The normal state points to the information of the last packet that has been marked as normal. The staging state stores metadata for the packets waiting for classification.

Next, we define a parameter T_m representing the maximum delay in the arrival of the next benign packet, which is equal to the TAL of the packet held in the normal state. The need for this extra variable arises because IEC 61850 only defines the allowed range for the configurations of GOOSE TAL and retransmission times (T_i's in Fig. 1) [1]. The actual settings vary across vendors; for example, TAL is often set equal to T_0 or $2 * T_0$. Moreover, whether different GOOSE messages can have different TAL values is also an implementation choice.

Staging-state maintenance relies on the following property:

Proposition 1. *If a GOOSE poisoning attack injects a set of packets, Q, then any $q \in Q$ will be between two benign packets that are, at the maximum, T_m time apart.*

Proof. The proof follows from GOOSE specifications. A publisher sends periodic messages either to retransmit a state or to notify subscribers of a new event concerning a `DatSet`. That behavior results in a time series of events similar to the example in Fig. 1. In this time series, the maximum delay between any two consecutive events is T_0, which needs to stay below T_m to keep the communication alive. Thus, since the attack needs to add extra packets, any given attack packet will be between two good packets, which are T_m or less apart. □

Algorithm 1 GOOSE Poisoning Detection

Input: st_{min}, st_{max}, sq_{min}, sq_{max}
Output: alerts
Initialize: $n \leftarrow (st_0, sq_0)$ ▷ `StNum` and `SqNum` of the first benign packet
$S \leftarrow \{(st_1, sq_1)\}$ ▷ `StNum` and `SqNum` of the second benign packet
$V \leftarrow \phi$ ▷ Detected attacks
$T_m \leftarrow n.\texttt{TAL}$ ▷ `TAL` of the last benign packet
$stateChanged \leftarrow \bot$

1: $p \leftarrow$ GETNEXTPACKET ▷ `StNum`, `SqNum`, and other fields
2: **schedule**(FLUSHSTAGING, T_m)
3: **while** p **do**
4: **if** NEXTEXPECTEDSQ(p, sq_{min}, sq_{max}) **then**
5: $S \cup \{p\}$
6: $stateChanged \leftarrow \bot$
7: **else if** NEXTEXPECTEDST(p, st_{min}, st_{max}) **then**
8: $S \cup \{p\}$
9: $stateChanged \leftarrow \top$
10: **else if** HIGHSQ(p) **or** HIGHST(p) **or** PRESENTINSTAGING(p) **or** ($stateChanged$ **and** PREVSTATERECURRED(p)) **then**
11: $V \cup \{p\}$
12: SENDALERT(p)
13: **break**
14: $p \leftarrow$ GETNEXTPACKET

15: **procedure** FLUSHSTAGING
16: **if** $|V| > 0$ **then** ▷ Attack(s) detected since the last activation
17: **return**
18: $n \leftarrow S[last - 1]$ ▷ Second to the last member
19: $S \leftarrow \{S[last]\}$ ▷ The last member
20: $T_m \leftarrow n.\texttt{TAL}$
21: **schedule**(FLUSHSTAGING, T_m)

We present GOOSE poisoning detection in Algorithm 1, which tracks the benign communication augmented with the normal and staging states. The algorithm is at the level of a `DatSet`. It requires as input parameters from a substation's configuration, including the minimum and maximum values for `StNum` and `SqNum`. At the setup, the normal state, n, refers to the first benign packet's metadata. The staging state, S, contains metadata for at least the next benign packet. V, an empty set, represents the information on attacks. Finally, T_m is set to the first benign packet's `TAL`, as noted previously.

The algorithm starts by scheduling FLUSHSTAGING to run at T_m time-points in the future. The activation of this procedure means that T_m time has elapsed since the last activation, and if there have been no attacks in this interval, the packets in S are benign. Therefore, the algorithm updates n, S, and T_m and schedules FLUSHSTAGING recursively (lines 18–21).

The algorithm's main body runs a loop until there are no packets to process, or an attack is detected. First, it checks whether a new packet, p, is valid, i.e., has the values for `StNum` and `SqNum` from one of the four valid states in Fig. 3. If it determines that the packet is valid, it appends p's information to S. The procedures NEXTEXPECTEDSQ and NEXTEXPECTEDST perform the checks (lines 4, 7), also taking into account the rollover scenarios. However, if p fails to satisfy the validity conditions, the algorithm then checks whether p relates to one of the violations (line 10). Functions HIGHSQ and HIGHST correspond to unusually high values of the counters, i.e., invalid states in Fig. 3. The third test, PRESENTINSTAGING, covers all the cases that correspond to the attacker's injection of the next expected packet. It returns *True* if a packet with the same `StNum` and `SqNum` values is present in S. Finally, PREVSTATERECURRED checks whether either the attacker or the publisher has inserted a new state, while the other is still retransmitting the old state. It compares p against the information in both n and S, to account for the situations in which the attack was injected concurrently with the transmission of the next benign packet.

On finding any violation, the algorithm appends p's metadata to V (line 11). SENDALERT (line 12) then triggers an alert that includes information such as source and destination addresses, `StNum` and `SqNum`, and timestamp. At that point, S contains the entire trace of packets that enabled the detection. A security analyst can subsequently act upon the alerts generated by the algorithm to bring the system back to a safe state and initialize n and S with new data.

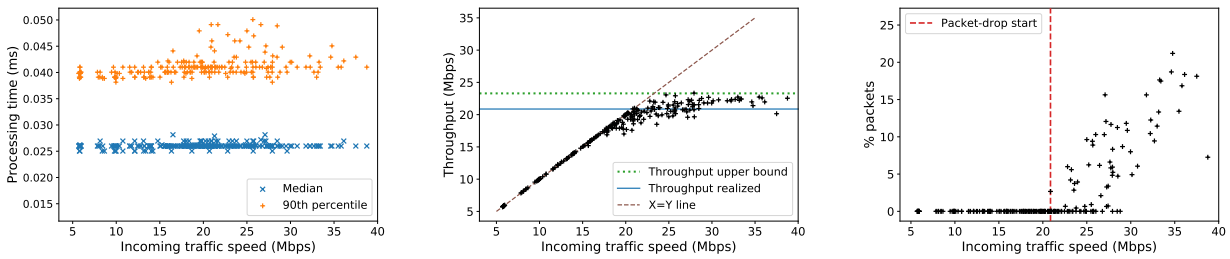
The algorithm has both time and space complexities of $\mathcal{O}(s)$, where s is the size of the staging state. Theoretically, s can be large, however, since attacks and state changes are rare, its expected value is low. Finally, the algorithm satisfies the following property:

Proposition 2. *GOOSE Poisoning Detection will detect all variants of the attack within T_m time units from the attack's occurrence, given the benign communications continue.*

Proof. Let $p_1 := (a, b)$ and $p_2 := (x, y)$ be the benign packets on the either side of an attack packet q, i.e., $t(p_1) \leq t(q) \leq t(p_2)$, where $t(.)$ is the arrival time. One of the following situations will happen:

- q injects a high `StNum` or high `SqNum` (invalid states in Fig. 3): The algorithm will detect this immediately (at $t(q)$).
- q takes one of the valid states: The arrival of p_2, depending on the values of (x, y), will result in either (1) a duplicate retransmission or (2) recurrence of an old state. Those two scenarios are detected at $t(p_2)$ by PRESENTINSTAGING and PREVSTATERECURRED, respectively. Since, $(t(p_2) - t(q)) \leq T_m$ (from Proposition 1), the time from occurrence to detection is always less than T_m. Finally, if the attacker injects multiple valid packets after q, they all remain in S until $t(p_2)$ and get detected. □

Proposition 2 provides a *sufficient* condition for the algorithm. Even if an attack involves a bad `TAL` to cause a timeout at the subscriber, the algorithm will detect it. However, if the publisher fails to send the next benign message within the T_m



(a) Processing time for GOOSE events. (b) Overall system throughput. (c) Packets dropped by Zeek.

Fig. 4: Performance of ED4GAP in processing the evaluation dataset on our experimental setup with four cores of CPU. Each marker denotes a run of an experiment in which all 14 traces were analyzed.

interval (i.e., the current `TAL`), the communication is deemed lost. In those cases, our algorithm will need to restart, which is a limitation, especially because how the devices are reset after a timeout is not consistent across different vendors.

IV. Experimental Evaluation

We conducted experiments to assess the detection accuracy and real-time performance of the ED4GAP. The experiments used an IEC 61850 security dataset collected from a simulated power system [17]. The devices in the simulated system normally send multicast packets every second to share their statuses. Anomalous behaviors corresponding to several benign power system disturbances and attacks are also present. Overall, the dataset consists of 14 traces amounting to about 86 megabytes and $523,853$ packets. GOOSE messages comprise about 87% of the traffic, whereas the remaining 13% includes conventional protocols, such as ARP, ICMP, DNS, and DHCP.

We implemented ED4GAP using the Zeek network security monitor [15]. Zeek provides an extensible platform with built-in analyzers for many protocols and a scripting language for security-policy development. To process GOOSE traffic, we used an analyzer proposed in [16]. It allowed us to parse Ethernet and GOOSE headers as well as the APDU to extract features essential for detection. Using those features, we implemented the GOOSE poisoning detection algorithm as a Zeek script. Finally, we utilized Zeek's logging infrastructure to generate logs of the GOOSE messages and alerts.

We installed Zeek (version Bro 2.5.4) on a virtual machine (VM) running Ubuntu 16.04.6 LTS with 2 GB of RAM and four virtual CPUs. The host computer had macOS version 10.15.4 on a 2.6 GHz quad-core Intel Core i7 with 8 GB of DDR3 RAM. We used `tcpreplay` to feed the existing traces to an interface that Zeek was monitoring.

Intrusion Detection Accuracy. Of the three types of attacks present in the dataset, message suppression (MS) and data manipulation (DM) are relevant for our evaluation. They inject packets with modified `StNum` or `SqNum`; DM attacks also manipulate data values. Therefore, we consider only the MS and DM attack scenarios for the security assessment.

The dataset, however, does not cover all forms of GOOSE poisoning. There are 24 possible variations, which consider all types of violations and benign changes. The variations correspond to the two fields $\{$`StNum`, `SqNum`$\}$, two options for the first attack packet $\{$a high value, the next value$\}$, three choices for how the attacker will follow up $\{$do nothing, retransmit, inject new state$\}$, and two benign changes $\{$retransmit, new state$\}$. The original traces cover only two of the 24 attack variations. Therefore, we generated new traces that rely on the dataset's normal trace to cover the remaining 22 test cases. We used Python's Scapy module [19] to inject manipulated packets at specific points. In each case, the number of injected attack packets was less than ten, which is very small compared to the normal traffic, e.g., 600 packets of every `DatSet`.

We fed all the traces, including those that we created, to ED4GAP. We compared the output alerts with the ground-truth information present at known points in the dataset. The system could detect all the attacks without any false positives and trigger alerts to identify source packets correctly.

Performance. In the current design, ED4GAP is non-blocking; it does analyze the traffic off mirror ports without interruption. Nevertheless, since GOOSE handles protection-related information, the security analysis must be fast. To confirm that ED4GAP meets that criterion, we evaluated its response time and throughput. In each experiment, we replayed all 14 traces and performed more than 200 trials.

ED4GAP showed an average per-packet response time of 0.06 ms. We computed the average response time by dividing the completion time by the total number of packets. Hence, we accounted for the network latency, waiting time, processing time, and log-writing time of all the protocols. Response time analysis thus indicates that the additional overhead due to ED4GAP is low. Next, we measured the processing time in handling only the GOOSE messages, which we defined as the time taken in packet analysis, attack detection, alert generation, and log/alert writing to the disk. Figure 4a depicts the GOOSE processing time as the speed of the input traffic increased. Both the median and the 90th percentile of the processing time (in ms) hold stable within the ranges $[0.025, 0.028]$ and $[0.038, 0.050]$, respectively. Those values are less than 2% of the GOOSE's most stringent transfer time. GOOSE processing time analysis thus provides empirical confirmation of the constant complexity of the poisoning detection approach.

Afterward, to test how ED4GAP behaves with high-speed input traffic, we measured its throughput, i.e., the rate of *successful* message processing. Our results showed that the throughput increased almost linearly with the input speed until it hit a plateau, as plotted in Figure 4b. Before reaching the plateau, however, the system started dropping packets. The maximum throughput realized without any packet drop was 20.87 Mbps. Packet drop incurred beyond that speed reached up to 25.95% of the total packets transmitted at around 35 Mbps speed. We show this effect in Figure 4c.

The packet drop reflects the capacity of our experimental setup. The theoretical limit of the throughput of our setup was 23.30 Mbps, which we measured by inputting the traffic directly to Zeek with no network interface in the middle (using the `zeek -r` command). The limit is depicted as a green horizontal dotted line in Figure 4b. However, when the traffic is read from a network interface, that upper bound would not be hit. The socket layer between the interface and Zeek would add delay. Those factors led to a throughput of 20.87 Mbps in our experiments, lower than the upper-bound.

Finally, in packets per second (pps), the throughput was $16,649$, since the evaluation dataset had an average packet size of 164.30 bytes. If all the GOOSE frames were of the maximum size of 1500 bytes, and considering an additional 22 bytes for the typical Ethernet plus VLAN overhead, a throughput of about 1797 pps would still be realized.

Architectural Considerations. In practice, ED4GAP will be deployed at a substation server. Although the performance of such a deployment can be better than our experimental setup (a VM on a typical workstation), one needs to consider several factors to speed up the execution, as we discuss next.

In general, the throughput will be better if more resources are available. Processing time, on the other hand, is a function of the complexity of the algorithm. Therefore, processing time may not decrease below some limit simply through access to additional resources. We found that ED4GAP was mainly CPU-bound. Therefore, to increase throughput, we can either add CPUs or use multiple similar Zeek workers running in parallel. To test that hypothesis, we separately collected the performance results on deployments with two and six CPU cores, in addition to the previously discussed results with four cores. As presented in Table I, the overall throughput of the system (with no packet drop) increased with the number of cores. Also shown in the table is the average percentage packet drop with its 95% confidence interval, which decreased when cores were added. That analysis shows that with fewer cores, the system got overloaded at a lower traffic-speed. Although Zeek's primary function is single-threaded, it uses multiple threads to perform secondary tasks, such as writing logs. In fact, we observed up to nine threads of the Zeek process. Therefore, the overall throughput improved when the number of cores increased.

A lower throughput indicates that Zeek was interrupted by other system processes. To analyze that effect, with a total of six cores, we pinned $c = \{1, 2, 3, 4\}$ cores exclusively to Zeek and measured the throughput and response time. Exclusion of c cores removed them from the Linux scheduler's pool, forcing it to consider only $6 - c$ cores for running other processes. Thus, Zeek could run uninterrupted on c cores. The results of that set of experiments, shown in Table II, reveal that the exclusive pinning helped improve the throughput compared to that of the unpinned setting. In summary, this analysis provides insights into system bottlenecks and can inform decisions that will improve performance.

TABLE I: Effect of increasing the number of CPU cores (n).

n	Throughput (Mbps)	Latency per packet (ms)	Avg % packet drop at speed $>$ throughput
2	12.30	0.10	25.93 ± 4.11
4	20.87	0.06	5.43 ± 1.30
6	21.15	0.06	2.58 ± 3.28

TABLE II: Effect of pinning c cores to Zeek; $n = 6$.

c	Throughput (Mbps)	Latency per packet (ms)
1	18.91	0.07
2	21.68	0.04
3	21.98	0.03
4	21.98	0.03

V. Practical Aspects and Limitations

Our analysis of bottlenecks and performance-improvement aspects has focused mainly on the effect of increasing compute resources. However, some network monitoring appliances implement various other techniques to accelerate packet processing. For instance, the R-Scope [20] network security appliance implements a set of packet-path optimization techniques that help accelerate the processing of Zeek-based security analytics, such as the ones we present in this paper. These high-performance techniques reaffirm the viability of running the presented system in real-time according to the protocol's timing requirements.

ED4GAP utilizes port mirroring to monitor the network, the overhead of which is negligible on the high-speed switching fabric [21]. Moreover, in practice, we can further minimize the impact on performance by selecting specific interfaces and using filters to send specific traffic to a port mirroring instance.

Currently, our threat model does not consider attacks that can compromise an IED to suppress/manipulate its behavior completely. Network-level detection of such attacks would require one to find anomalies by taking into account both the cyber and physical properties of GOOSE messages. An extension of our system to incorporate such models is possible.

Finally, since the poisoning detection relies on storing traffic metadata in a staging state, a memory attack causing a denial-of-service (DoS) of ED4GAP is possible. Such an attack would need to flood the network with a large number of packets in an interval smaller than T_m. In this paper, we do not aim to detect flooding-based DoS. However, some existing systems can help mitigate those attacks [22], [23].

VI. Related Work

Network-based intrusion detection is an effective strategy for substation security, including protection against threats

to GOOSE communication. It is the basis of many related research efforts, e.g., [22], [24]–[26]. In particular, the negative impact of GOOSE poisoning on the availability of substation devices is presented in [13], [14]. Papers that propose detectors for such attacks include [16], [27]. However, those solutions cover only a limited number of attack variants. In contrast, the presented solution relies on the GOOSE protocol specifications to detect the attack in all variations, including cases in which the attacker closely follows the normal pattern. Moreover, we uniquely provide a systematic analysis of the performance to demonstrate that the system can achieve a low overhead that is suitable for GOOSE communication. Papers that, like ours, incorporate Zeek for SCADA network monitoring and security analysis include [16], [24], [25]. Finally, our experiments utilize the GOOSE analyzer proposed in [16] and the substation dataset developed in [17].

VII. Conclusion

We introduce in this paper a network-level system, ED4GAP, dedicated to detecting GOOSE poisoning attacks in modern substations. ED4GAP relies on the extraction of features from substation traffic, analysis of protocol specifications, and a comprehensive examination of communication properties to detect violations and generate alerts. We have implemented our system using the Zeek platform and devised a systematic approach to evaluate its performance. Our results show that the system can analyze traffic and accurately detect the poisoning attacks concurrently to the data transmission. Thus, it is suitable for the context of high-speed communication within a substation. The next steps will be to extend the security analysis to evaluate both the cyber and physical properties of the control traffic and test the presented system in a live substation environment to study the effect of network conditions, such as latency, jitter, and loss on its performance.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments, and Jenny Applequist for her editorial assistance. This material is based upon work supported in part by the Department of Energy under Award Number DE-OE0000780 and in part by Helmholtz Programm Energieeffizienz, Materialien und Ressourcen (34.15.01), Kastel BMBF Projekt Sicherheit kritischer Infrastrukturen, BMBF Energiesystem 2050 and the Karlsruhe House of Young Scientists. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

References

[1] IEC 61850-8-1:2011+AMD1:2020 CSV communication networks and systems for power utility automation: Part 8-1. 2020. [Online]. Available: https://webstore.iec.ch/publication/66585

[2] IEC TS 62351-6:2007 data and communications security: Part 6: Security for IEC 61850. 2007. [Online]. Available: https://webstore.iec.ch/publication/6909

[3] F. Hohlbaum, M. Braendle, and F. Alvarez, "Cyber security practical considerations for implementing IEC 62351," in *PAC World Conf.*, 2010, pp. 1–8.

[4] M. G. da Silveira and P. H. Franco, "IEC 61850 network cybersecurity: Mitigating GOOSE message vulnerabilities," in *Proc. 6th Annual PAC World Americas Conf.*, 2019, pp. 1–9.

[5] M. El Hariri, T. A. Youssef, and O. A. Mohammed, "On the implementation of the IEC 61850 standard: Will different manufacturer devices behave similarly under identical conditions?" *Electronics*, vol. 5, no. 4:85, pp. 1–13, 2016.

[6] W. Fangfang, W. Huazhong, C. Dongqing, and P. Yong, "Substation communication security research based on hybrid encryption of DES and RSA," in *Proc. 2013 9th Int. Conf. on Intelligent Inform. Hiding and Multimedia Signal Processing*, 2013, pp. 437–441.

[7] A. P. Premnath, J. Jo, and Y. Kim, "Application of NTRU cryptographic algorithm for SCADA security," in *Proc. 2014 11th Int. Conf. on Inform. Technology: New Generations*, 2014, pp. 341–346.

[8] S. M. Farooq, S. M. S. Hussain, and T. S. Ustun, "Performance evaluation and analysis of IEC 62351-6 probabilistic signature scheme for securing GOOSE messages," *IEEE Access*, vol. 7, pp. 32 343–32 351.

[9] K. Boakye-Boateng and A. H. Lashkari, "Securing GOOSE: The return of one-time pads," in *Proc. 2019 Int. Carnahan Conf. on Security Technology*, 2019, pp. 1–8.

[10] G. Elbez, H. B. Keller, and V. Hagenmeyer, "Authentication of GOOSE messages under timing constraints in IEC 61850 substations," in *Proc. 6th ICS-CSR Symposium*, 2019, pp. 137–143.

[11] D. Ishchenko and R. Nuqui, "Secure communication of intelligent electronic devices in digital substations," in *Proc. 2018 IEEE/PES Transmission and Distribution Conf. and Exposition*, 2018, pp. 1–5.

[12] E. Esiner, D. Mashima, B. Chen, Z. Kalbarczyk, and D. Nicol, "F-Pro: a fast and flexible provenance-aware message authentication scheme for smart grid," in *Proc. 2019 SmartGridComm*, 2019, pp. 1–7.

[13] J. Hoyos, M. Dehus, and T. X. Brown, "Exploiting the GOOSE protocol: A practical attack on cyber-infrastructure," in *Proc. 2012 IEEE Globecom Workshops*, 2012, pp. 1508–1513.

[14] N. Kush, E. Ahmed, M. Branagan, and E. Foo, "Poisoned GOOSE: Exploiting the GOOSE protocol," in *Proc. 12th Australasian Inform. Security Conf.*, 2014, pp. 17–22.

[15] The Zeek network security monitor. 2020. [Online]. Available: https://www.zeek.org/

[16] M. Kabir-Querrec, "Cyber security of smart-grid control systems: Intrusion detection in IEC 61850 communication networks," Ph.D. dissertation, Université Grenoble Alpes, 2017.

[17] P. P. Biswas, H. C. Tan, Q. Zhu, Y. Li, D. Mashima, and B. Chen, "A synthesized dataset for cybersecurity study of IEC 61850 based substation," in *Proc. 2019 IEEE SmartGridComm*, 2019, pp. 1–7.

[18] A. Valdes, C. Hang, P. Panumpabi, N. Vaidya, C. Drew, and D. Ischenko, "Design and simulation of fast substation protection in IEC 61850 environments," in *2015 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems*, 2015, pp. 1–6.

[19] Python Scapy. 2020. [Online]. Available: https://scapy.net/

[20] J. Ros-Giralt, A. Commike, P. Cullen, and R. Lethin, "Algorithms and data structures to accelerate network analysis," *Future Generation Computer Systems*, vol. 86, pp. 535–545, 2018.

[21] Cisco catalyst SPAN configuration example. 2019. [Online]. Available: https://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/10570-41.html

[22] U. K. Premaratne, J. Samarabandu, T. S. Sidhu, R. Beresh, and J. Tan, "An intrusion detection system for IEC61850 automated substations," *IEEE Trans. on Power Delivery*, vol. 25, no. 4, pp. 2376–2383, 2010.

[23] Q. Yang, W. Hao, L. Ge, W. Ruan, and F. Chi, "Farima model-based communication traffic anomaly detection in intelligent electric power substations," *IET Cyber-Physical Systems: Theory Applications*, vol. 4, no. 1, pp. 22–29, 2019.

[24] W. Ren, T. Yardley, and K. Nahrstedt, "EDMAND: Edge-based multi-level anomaly detection for SCADA networks," in *Proc. 2018 IEEE SmartGridComm*, 2018, pp. 1–7.

[25] V. Coughlin, C. Rubio-Medrano, Z. Zhao, and G. Ahn, "EDSGuard: Enforcing network security requirements for energy delivery systems," in *Proc. 2018 IEEE SmartGridComm*, 2018, pp. 1–6.

[26] A. Albarakati, C. Robillard, M. Karanfil, M. Kassouf, R. Hadjidj, M. Debbabi, and A. Youssef, "Security monitoring of IEC 61850 substations using IEC 62351-7 network and system management," in *Proc. 2019 IEEE SmartGridComm*, 2019, pp. 1–7.

[27] J. Hong, C. Liu, and M. Govindarasu, "Detection of cyber intrusions using network-based multicast messages for substation automation," in *Proc. Innovative Smart Grid Technologies*, 2014, pp. 1–5.