Practical Application Layer Emulation in Industrial Control System Honeypots

Kyle Girtz, Barry Mullins, Mason Rice, Juan Lopez

Chapter 5

PRACTICAL APPLICATION LAYER EMULATION IN INDUSTRIAL CONTROL SYSTEM HONEYPOTS

Kyle Girtz, Barry Mullins, Mason Rice and Juan Lopez

Abstract Attacks on industrial control systems and critical infrastructure assets are on the rise. These systems are at risk due to outdated technology and *ad hoc* security measures. As a result, honeypots are often deployed to collect information about malicious intrusions and exploitation techniques. While virtual honeypots mitigate the excessive cost of hardware-replicated honeypots, they often suffer from a lack of authenticity. In addition, honeypots utilizing a proxy to a live programmable logic controller suffer from performance bottlenecks and limited scalability. This chapter describes an enhanced, application layer emulator that addresses both limitations. The emulator combines protocol-agnostic replay with dynamic updating via a proxy to produce a device that is easily integrated into existing honeypot frameworks.

Keywords: Industrial control systems, honeypot, emulator, proxy

1. Introduction

Technological advancements on a societal scale require a stable underlying critical infrastructure that generates and distributes electricity, gas, water, communications, commercial goods and other necessities. In the United States and other nations, critical infrastructure assets are monitored and managed by industrial control systems. Historically, industrial control systems were isolated and designed for robustness rather than security. Today, security requirements rival availability as industrial control systems have become increasingly interconnected, exposed to the Internet and accessible to attackers.

The programmable logic controller (PLC), a common industrial control device, is particularly important to securing industrial control systems. These devices contain custom programs that support data collection and actuator control. Modern malware, such as Stuxnet, have successfully compromised pro-

grammable logic controllers with destructive results [2]. To complicate matters, the need for uninterrupted critical infrastructure services makes it very difficult to update or patch industrial control systems in the traditional manner. Although security measures are desperately needed, they must be implemented appropriately. To accomplish this, network administrators and security experts need to know exactly how to best defend industrial control networks.

Honeypots are a deception-based technology that is commonly employed for network state detection, threat analysis and data collection. A honeypot is a bait device added to a network that attracts attackers and collects suspicious traffic [5]. This chapter describes an enhanced production honeypot configuration for industrial control systems that incorporates secondary emulation at the application layer. The emulator combines protocol-agnostic replay with dynamic updating via a proxy to produce a device that is easily integrated into existing honeypot frameworks.

1.1 Honeypots

A honeypot is a passive device designed for information gathering [5]. A physical honeypot is a hardware duplicate of the target system. A virtual honeypot is a software simulation designed to behave in a similar manner as a target system.

The degree of honeypot interaction is characterized as high or low depending on how much of the target system is replicated by the honeypot. A high interaction honeypot is a full computer system that operates with complete functionality [6]. Leveraging physical hardware or a virtual machine (VM), such a honeypot provides real services and genuine interactions to an attacker. In contrast, a low interaction honeypot does not provide an entire, functional computer system for attacker interactions. Instead, it can only emulate specific services, network stacks or other aspects of a real system [6].

Honeypots are evaluated based on three operational characteristics: (i) performance; (ii) authenticity; and (iii) security [6]. Performance refers to the ability of a honeypot to handle heavy traffic loads and project multiple virtual devices simultaneously. Authenticity indicates how closely a honeypot mimics the functionality of a real device. Security describes the vulnerability of a honeypot in the event an intruder obtains access to the honeypot and pivots to real devices on the network.

1.2 Industrial Control System Honeypots

Due to the substantial differences between traditional information technology hardware and industrial control devices such as programmable logic controllers, deploying an industrial control system honeypot can be a challenging task. High interaction honeypots are difficult to scale because a single programmable logic controller can cost thousands of dollars; in addition, these devices are rarely virtualized with success. Low interaction honeypots have

arduous configuration processes and lack authenticity due to the difficulty of virtualizing proprietary hardware and networking protocols.

The CryPLH research effort [1] has attempted to create a custom industrial control system honeypot using an Ubuntu virtual machine. The goal is to create an authentic honeypot that is easy to configure and that can be modified to emulate similar programmable logic controllers relatively quickly. The design incorporates a stripped down virtual machine configured to look exactly like a Siemens Simatic 300(1) programmable logic controller. Using `iptables` as a firewall/filter, the virtual machine is able to provide a variety of services, including HTTP and HTTPS, SNMP and the ISO-TSAP protocol used by Siemens in its STEP7 programming software. The design is flexible, but it requires the manual configuration of each service provided by the honeypot.

A similar design based on custom Linux configurations is the highly portable industrial control system honeypot created by Jaromin [3] using a Gumstix device. Like CryPLH, this honeypot provides chosen services using manually-configured firewall rules and custom scripts. The honeypot emulates a single device – a Koyo DirectLOGIC 405 programmable logic controller with HTTP and Modbus services. Although the honeypot performs well, it has limited applicability because the Gumstix hardware is restricted to a single programmable logic controller configuration and each service must be configured manually.

1.3 Current Technology

Two recent industrial control system honeypot frameworks, Honeyd+ and ScriptGenE, provide partial solutions using opposite approaches.

The Honeyd system is not a single honeypot, but a framework for creating virtual networks of honeypots [5]. It enables users to create arbitrarily many virtual, low interaction honeypots and virtually network them together to consume unused IP space in a real network. In addition, Honeyd offers great flexibility via service scripts, allowing the virtual honeypots to run any service or protocol desired by a user. To increase authenticity, Honeyd also projects operating systems using signatures from the same databases referenced by scanning tools such as `nmap`. To enhance flexibility and authenticity, Honeyd allows users to install subsystems and user-specified external applications that run as components of a honeypot [6].

Winn et al. [8, 9] have extended Honeyd using a proxy to provide enhanced authenticity. The resulting Honeyd+ is designed to be an inexpensive production-level industrial control system honeypot framework. It can be deployed on a Linux Raspberry Pi and configured to proxy to a physical programmable logic controller at a remote location. With the Honeyd foundation, Honeyd+ enables honeypots to be deployed at multiple geographical locations, each honeypot emulating the same back-end programmable logic controller that is queried for requests at the application layer. The Honeyd+ system also improves on Honeyd by adding a search-and-replace function to the web pages retrieved by the proxy. This ensures that an attacker cannot identify a honeypot by a discrepancy in its IP or MAC addresses.

While Honeyd+ forwards application traffic to a real programmable logic controller, ScriptGen creates a new Honeyd service script from an observed network trace [4]. ScriptGen uses state machines to determine the structure of a traffic dialog without any knowledge of the protocol or its implementations on a server or client. The state machine can be simplified and converted into a Python script usable by Honeyd. Protocol agnosticism enables ScriptGen to automatically replay protocols that may be proprietary or simply unexpected.

ScriptGenE, an extension of ScriptGen created by Warner [7], can handle difficult cases such as session looping and default responses during replay. The protocol-agnostic design is ideal for industrial control systems that use proprietary protocols. As an extension of ScriptGen, the ScriptGenE framework constructs a protocol tree (p-tree) as a finite state machine. The p-tree can be converted to a Honeyd script or ScriptGenE can access it directly to replay the conversation as a subsystem of Honeyd or as an independent deployment.

2. Emulator Methodology

This section describes the emulator design considerations and features.

2.1 Design Considerations

While Honeyd+ is a cost-effective honeypot framework, its primary weakness is scalability with regard to performance. All application layer traffic is forwarded to a programmable logic controller to ensure authenticity. The backend programmable logic controller is easily overloaded by substantial traffic and large numbers of deployed honeypots [9]. Programmable logic controllers are not typically designed for optimal networking performance.

Replacing the programmable logic controller with an application layer emulator like ScriptGenE is a reasonable solution. However, authenticity suffers due to emulator capabilities, or, in the case of ScriptGenE, the extent of the training data. If the honeypot forwards a request that the emulator cannot handle, the response is generally less authentic than if the response had come from a real device.

The ideal emulator solution alleviates programmable logic controller load without compromising authenticity. The goal of this research was to create an enhanced emulator that has this characteristic. ScriptGenE, a replay emulator, which provides the foundation, is extended to include dynamic updating via a proxy. Figure 1 shows how the emulator may be inserted into an existing Honeyd+ network to reduce traffic load on a programmable logic controller.

2.2 Proxy and Update Features

The complete ScriptGenE suite includes Python tools for automatically generating generic p-trees from observed traffic and replaying the trees. During a replay, ScriptGenE maintains a context node in the preloaded p-tree. Incoming client messages are matched against the outgoing edge of the current context

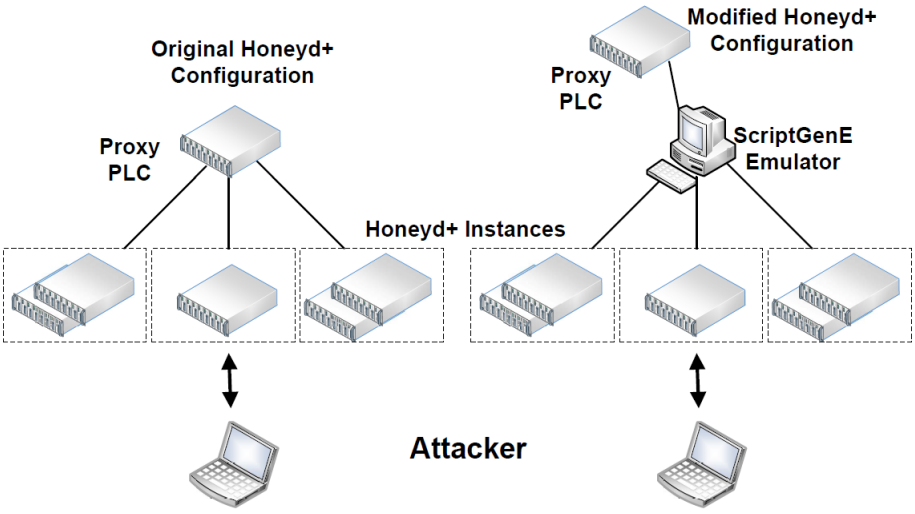

Figure 1. Honeyd+ configuration modification.

node. A correct match indicates that the child node along the edge holds the appropriate server response. If no match is found, a backtracking algorithm can search the tree for a potential response in a different context. This enables the emulator to handle session looping. A default error message is sent if no match is found in the entire tree.

The design assumes a properly-built tree and extends the replay functionality exclusively. The enhanced emulator substitutes the default error message with a proxy-and-update process when an unrecognized client message is received. The process incorporates the following steps:

- Synchronize the conversation context with the programmable logic controller according to the chosen context algorithm.
- Replace the environmental information in the unrecognized packet.
- Send the unrecognized packet to the programmable logic controller.
- Collect the programmable logic controller response.
- Replace the environmental information in the programmable logic controller response.
- Create a new node in the p-tree to store the response.
- Change the current p-tree context to the new node and send the server response to the client.
- Handle the programmable logic controller connection according to the chosen context algorithm.

Some of these steps may be omitted or reordered depending on the context synchronization algorithm that is employed. Synchronization is an interesting problem that is discussed in the next section. The other steps are:

- **Environmental Information Replacement:** Environmental information, such as IP addresses, port numbers and host names, vary for hosts, even in the case of identical conversation content. P-tree generation identifies these items for replay and replacement as necessary. However, an unrecognized client message may contain environmental information. Because this information is not in the p-tree, an exhaustive search for all possible environment fields is necessary before the message can be forwarded to the programmable logic controller. Similarly, the programmable logic controller response must be searched and revised before it is sent to the client.
- **Programmable Logic Controller Interaction:** The emulator creates a separate thread for each new connection so that the emulator can handle concurrent client connections. Each thread maintains a unique proxy connection to the programmable logic controller to ensure the correct context for the conversation. If errors occur in a programmable logic controller connection, the proxy is abandoned and the emulator reverts to a default error message.
- **Protocol Tree Update:** A new child node is added to the current context in the p-tree. The connecting edge contains the unrecognized client message and the new node contains the programmable logic controller response. Future requests of this type are replayed directly instead of being proxied again. Unfortunately, this solution is temporary; the updated tree is not saved when the emulator is terminated. A p-tree is a generic structure constructed from multiple traces containing the same kind of traffic. Updates add a single, real message instance to an abstract p-tree. Bytes that vary in the messages of the new type are not detected. It is safer to record all proxy traffic from outside the emulator and build a new p-tree for future emulation.
- **New Response to Client:** The new server response is forwarded to the client in place of a default error message. This enhances the authenticity of the emulator.

2.3 Synchronization Algorithms

One of the challenges in updating emulation capabilities using a proxy is to synchronize conversation context with the programmable logic controller. The problem arises when an unknown client request occurs while the replay state is deep in the p-tree. In order for the programmable logic controller to return the appropriate response to this new request, it must be caught up on the current context of the conversation. Synchronizing the client conversation with the programmable logic controller requires sending each client message

along the path from the current p-tree node up to the root in reverse order. These messages can be sent all at once or individually as they are received from the client or in any combination. Note that all the messages should be sent to guarantee the correct context. Failure to transfer the full context may not be a problem in every case, but it is difficult to know when it is necessary without detailed prior knowledge of the device and protocol functionality.

Two naive synchronization approaches are available. In the first approach, synchronization occurs entirely on demand (catch up). The programmable logic controller is ignored until an unrecognized request has to be forwarded. At this point, full synchronization occurs, the request is sent and the connection is terminated until another unrecognized request arrives and the process restarts. While this approach is ideal with regard to the performance of the programmable logic controller, it may unnecessarily resend synchronization traffic that could lead to significant delays from the attacker's perspective if the tree is very large.

The second (opposite) approach is to maintain the exact mirrored context with the programmable logic controller at all times (lockstep). All the received traffic is forwarded immediately so that the programmable logic controller is always ready to respond to an unrecognized request. This approach is ideal from the perspective of the emulator because delays are minimized and the context is never a problem. However, the approach does not alleviate the traffic load on the back-end programmable logic controller.

An understanding of the two extremes in synchronization is useful to develop an intermediate solution that alleviates programmable logic controller load while enhancing emulator efficiency. One possibility is to perform synchronization on demand for the first unrecognized request and then maintain the context until the connection ends (latelock).

An improvement on latelock is to maintain the context only as long as unrecognized requests are received (templock). The proxy connection is terminated at the first sign of a recognizable request for which the programmable logic controller is not needed.

An altogether different approach is to maintain the synchronized context whenever the client conversation is below a specified depth in the p-tree (triggerlock). This enables the emulator to avoid large synchronization delays without overwhelming the programmable logic controller.

Figure 2 provides a visual comparison of all the message-triggered algorithms. The bar for each algorithm denotes the duration of the proxy connection. The left side of a bar represents the initial synchronization, upon which the context is maintained; the right side of the bar denotes the point at which the proxy connection is closed.

A final consideration is the "setup phase" of many connections. The setup phases tend to be linear in a p-tree. After a session is established, a request may be sent in any order (corresponding to the branches of the p-tree). It is possible that minimal synchronization may be more efficient for some protocols. Only the linear path from the root to the first branch is synchronized. This

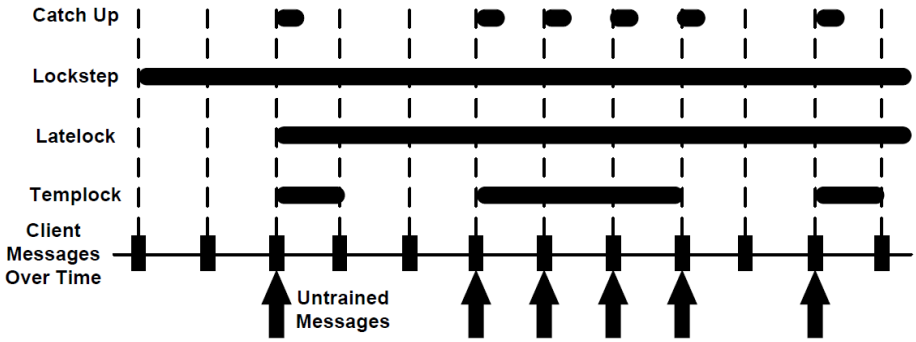

Figure 2. Comparison of context synchronization algorithms.

allows the proxy connection to synchronize much more quickly regardless of the current conversation context. This optimization may be combined with any of the other approaches that have a synchronization phase.

The correct choice of algorithm depends heavily on the nature of the protocol being emulated. For example, the catch up algorithm performs ideally on a stateless protocol, but it performs poorly on complex protocols with long conversations. Thorough protocol knowledge is required to determine the algorithm that should be chosen. Even then, the actual performance depends on the traffic that is received.

2.4 Design Limitations

The enhanced emulator has some limitations. Like its ScriptGenE foundation, it can only handle IPv4 addresses and TCP protocols. Encrypted protocols are not supported. While ScriptGenE is intended to be automated and fully protocol agnostic, the current enhancements require some manual configuration. If a protocol uses a field that is consistent across all packets during a connection (e.g., session ID), the current p-tree generation algorithm fails to identify the global field. New responses from the programmable logic controller may have different values from what are expected by a client. The enhanced emulator can replace these values to ensure authenticity, but the location and length of the field must be manually provided in the current software iteration.

3. Emulator Evaluation

The primary design goals for the emulator were to reduce the load on the back-end programmable logic controller in Honeyd+ networks and to enhance the authenticity in the presence of p-tree deficiencies. An additional goal was to determine the best choice of synchronization algorithm for real protocols. The evaluation of the emulator involved testing the design on diverse tasks and protocols to determine the performance with regard to load reduction and authenticity.

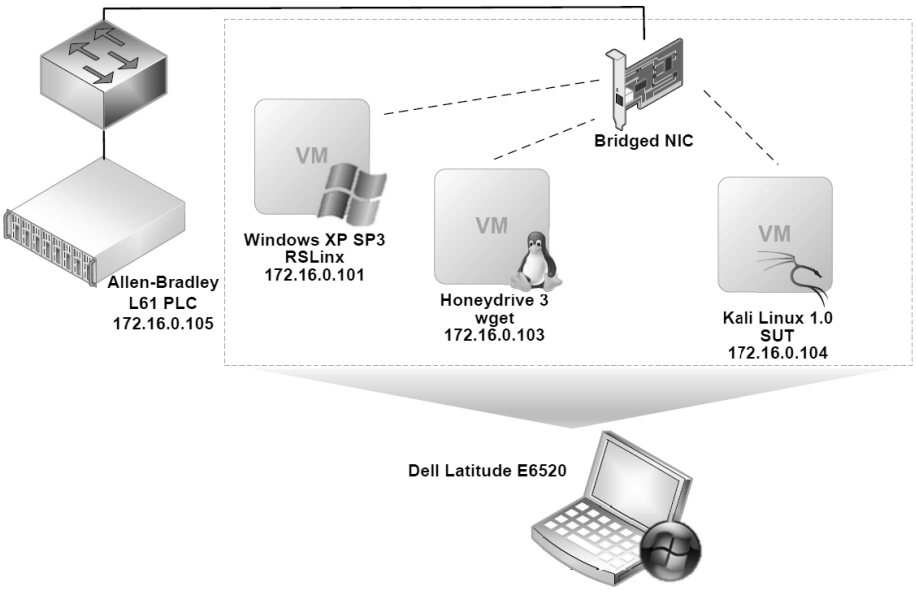

Figure 3. Experimental network configuration.

3.1 Experimental Design

The test environment consisted of an Allen-Bradley ControlLogix L61 programmable logic controller with a 1756-EWEB module connected via a private local network to a Dell Latitude E6520 laptop through a Cisco SG 100D-08 switch. The laptop hosted three virtual machines (i.e., two clients and the system under test). A Windows XP virtual machine running RSLinx acted as a browsing client for the EtherNet/IP protocol while a Honeydrive 3.0 virtual machine ran `wget` to provide an HTTP client. The system under test executed on a Kali Linux 1.0 virtual machine and acted as the honeypot in the network. Specific tasks involving the HTTP and EtherNet/IP protocols were chosen and automated using the SikuliX GUI automation software. All task coordination, emulator creation and traffic collection occurred on the Kali virtual machine to facilitate the automation of the experiments. Figure 3 shows the network configuration.

Each experimental task involved a chosen protocol (HTTP or EtherNet/IP), synchronization algorithm and modified p-tree. The synchronization algorithms evaluated were latelock, templock, minimal sync templock, triggerlock and no proxy usage (off mode), which provided a baseline. The algorithms were chosen because they are most likely to be deployed in a live system. The modified p-trees were generated from a baseline tree that accurately replayed the chosen task for its protocol. Modifications involved removing one random, non-root node from the baseline p-tree and deleting the descendants of the chosen node. The full experiment randomly ordered the tasks corresponding to all combina-

Figure 4. Success and failure in the `wget` results.

tions of the two protocols, five algorithms and seven p-trees (one baseline and six modified).

- **Programmable Logic Controller Load Testing:** For each task, two packet capture files were generated, one for the client connection and the other for the proxy connection. Each capture was filtered to determine the number of data-bearing messages sent by the "client" (the emulator is the client on the proxy side). The ratio of proxy messages to actual client messages reveals the amount of received traffic that had to be forwarded by the emulator to the programmable logic controller. A ratio less than 100% indicates a load reduction on the programmable logic controller in a Honeyd+ network. Comparison of the load metrics for the synchronization algorithms reveals the relative performances for each protocol and p-tree.

- **Authenticity Testing:** Each task was declared a success or failure by the GUI automation software. SikuliX searched the client screen for the required images and decided whether or not a task was completed successfully. Figure 4 compares the `wget` results for successful and failed tasks. A failed task was unable to provide all the needed files during download.

  Figure 5 shows the results for RSLinx browsing of the test network. The programmable logic controller shows up at `172.16.0.105` while the emulator at `172.16.0.104` only shows up when it can successfully pass the experimental task. The overall success rate when the proxy is used can be compared against the rate for default error messages. If the overall success rate is higher for the proxy, it can be concluded that proxy updates enhance the overall authenticity of the emulator.

3.2 Limitations

While the ultimate goal of application layer emulation is automated protocol agnostic replay, the experimental evaluation only provides an indication of

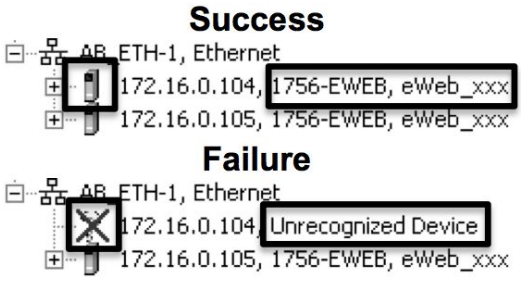

Figure 5. Success and failure in the RSLinx results.

emulator performance for the chosen protocols and configurations. Timing considerations, including variances in replaying recorded conversations and delays observed by the client during programmable logic controller synchronization, were not considered in the evaluation.

4. Experimental Results

All the evaluation tasks completed without errors. The programmable logic controller load test indicates that the emulator can reduce or maintain the same programmable logic controller load in all cases, except for one synchronization algorithm. The authenticity tests reveal that task success rates increase when the proxy updates replace default error messages. In the case of general proxy analysis, the synchronization algorithms are not distinguishable. The proxy was either considered to be on or off.

Note that the performance metrics generated by this experiment reflect the worst-case emulator performance. Each task was performed once for each emulation instance so that all untrained traffic required forwarding. In a live implementation, the tasks would likely be repeated over the life of a single emulation instance so that untrained requests would be proxied the first time and subsequently replayed from the updated p-tree.

4.1 Load Testing

Figure 6 shows the programmable logic controller load data for the HTTP protocol. Because each modified tree lacked exactly one random node and HTTP sends each individual request over a separate connection, the forwarding rate percentages are consistently 1.5%, a drastic improvement over 100%.

In some cases, two messages were received at a connection, doubling the forwarding rate. Even the worst case of 3% is much lower than the 100% target, indicating a large reduction in programmable logic controller traffic on a Honeyd+ network.

Figure 7 shows the programmable logic controller load data for the EtherNet/IP protocol. EtherNet/IP tree modifications varied widely due to the small number of branches in the p-tree. Almost all the client traffic was proxied or

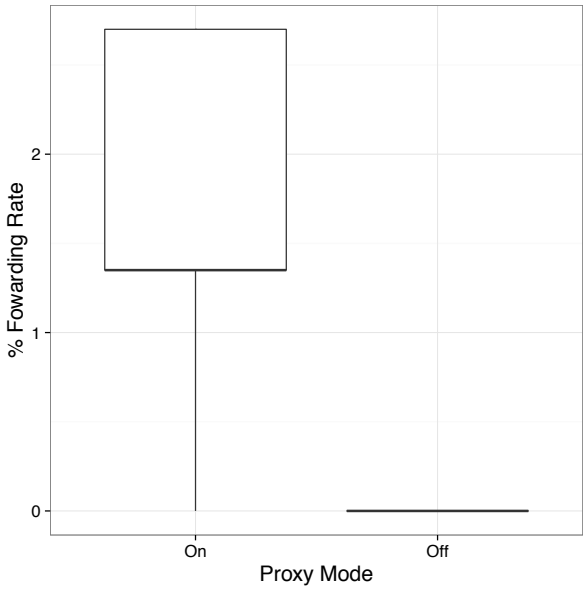

Figure 6. HTTP forwarding rates for tasks with and without the proxy.

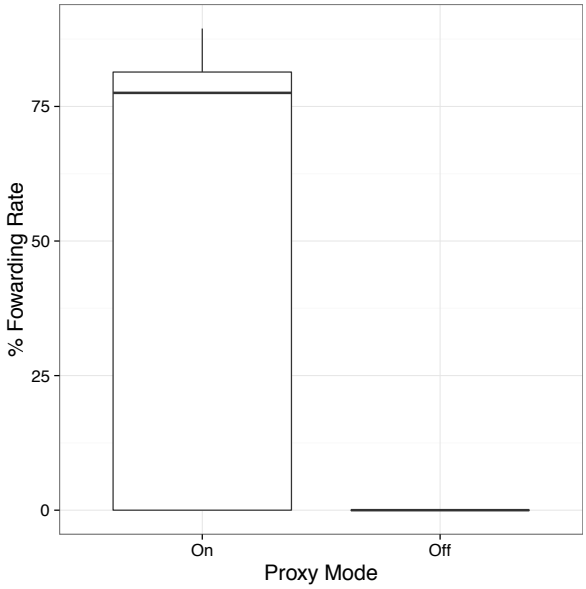

Figure 7. EtherNet/IP forwarding rates for tasks with and without the proxy.

sent to the programmable logic controller during context synchronization. This results in the high average forwarding rate seen in Figure 7. The values range from 0% to 90% and depend heavily on the extent of the p-tree modifications. Because of the linear structure, a missing node may disconnect a very small

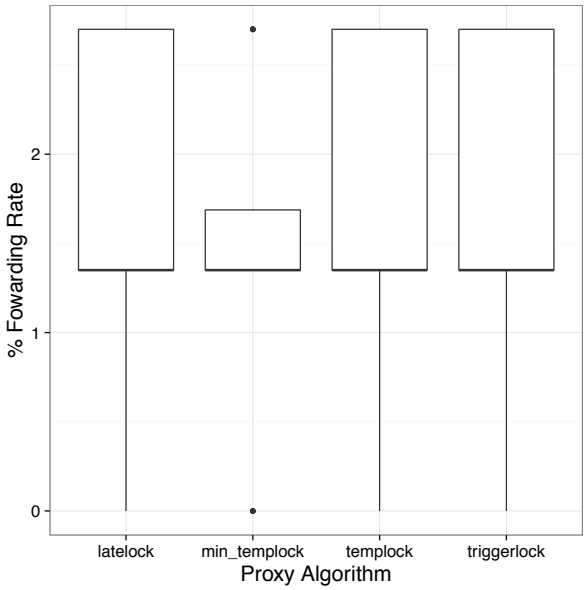

Figure 8. HTTP forwarding rates for the proxy algorithms.

portion or a large portion of the tree. Although the individual results vary, the aggregated data shows a reduction in traffic forwarded to the programmable logic controller.

4.2 Synchronization Algorithm Comparison

Because HTTP is a stateless protocol, HTTP tasks do not reveal useful information about the performance of synchronization algorithms. This is illustrated in Figure 8, where the algorithms have nearly identical performance.

Figure 9 shows that the triggerlock algorithm performs worse than the other algorithms for EtherNet/IP. This can be attributed to the session looping within the task, which causes the triggerlock algorithm to connect and disconnect from the programmable logic controller when the proxy is not always necessary. This occurs because the triggerlock algorithm creates and closes connections based on the depth in the tree instead of an actual need for the proxy.

The other algorithms exhibit very similar performance. This suggests that the experimental tasks did not provide sufficient variability to distinguish between the very similar algorithm behaviors. Because this variability is tied directly to the structure of the p-tree, entirely new protocols are necessary to extract the subtleties. The investigation of this issue is a component of future research.

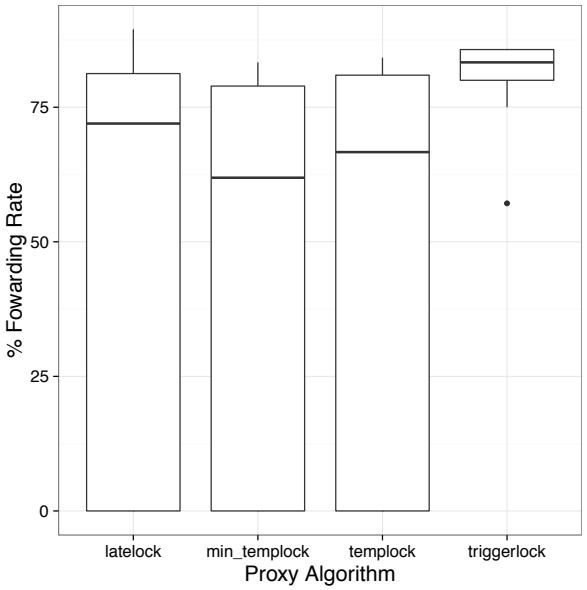

Figure 9. EtherNet/IP forwarding rates for the proxy algorithms.

Table 1. Pass rate results (%).

Protocol	Proxy	Expt. 1	Expt. 2	Expt. 3	Mean
HTTP	Off	0.00	0.00	0.00	0.00
	On	79.17	100.00	91.67	90.28
ENIP	Off	33.33	33.33	33.33	33.33
	On	48.00	45.83	48.94	47.59

4.3 Authenticity Testing

Table 1 shows the pass/fail results for all the modified-tree tasks in terms of percentage passing rates. With the proxy, the emulator was able to successfully complete nearly all the HTTP tasks while the default error messages failed to complete any task. This shows that the proxy can correctly supplement the p-tree during replays in order to provide authentic results. The results of the EtherNet/IP tests are less than ideal, with just 14% average improvement.

Some modified trees failed for all the synchronization algorithms. This indicates some tree modifications negatively impact the ability of the emulator to update correctly using the proxy connection. Manual inspection revealed that the emulator incorrectly expects more data when encountering a short message for the first time. Waiting for a second message causes the emulator to get out of sync with the client and reply to each request with the wrong response. All the other trees had 100% pass rates for all the algorithms.

It is also important to note that some minor modifications to the EtherNet/IP tree were not enough for the default error messages to cause the tasks to fail. This is a testament to the robustness of the foundational ScriptGenE replay framework.

5. Conclusions

The ScriptGenE framework provides a powerful tool for automated protocol replay. The improvements described in this chapter make ScriptGenE a practical application layer emulator. In a Honeyd+ network, adding the emulator as an intermediate to the programmable logic controller target can reduce the network load on the programmable logic controller to a degree that depends on the protocol. In the case of HTTP, the load reduction can be very large with thorough training data.

The problem when adding the emulator to a Honeyd+ network is that emulated traffic does not always match the programmable logic controller behavior exactly, as demonstrated by the occasional task failures. However, experiments reveal that proxy updates provide improved authenticity over the original default error message responses.

The experimental tests do not conclusively identify the proxy algorithm that provides the best performance. The algorithms are similar enough in that very specific tests are required to distinguish their performance gains and losses. However, it is clear that the triggerlock algorithm should be used carefully because it can actually increase the amount of traffic sent to the programmable logic controller.

Additional research and development is necessary to make ScriptGenE a practical product. The manual configuration of session IDs in protocols could be replaced by full protocol-agnostic automation. Further testing is needed to determine the optimal context synchronization algorithm. Additionally, the emulator has not been tested to determine if p-tree updates can provide performance improvements over time because fewer requests need to be proxied. The resolution of these issues and a more streamlined design would result in a product that can be added effortlessly to an existing Honeyd+ deployment to provide increased awareness of malicious intrusions into critical infrastructure assets.

Note that the views expressed in this chapter are those of the authors and do not reflect the official policy or position of the U.S. Air Force, U.S. Army, U.S. Department of Defense or U.S. Government.

References

[1] D. Buza, F. Juhasz, G. Miru, M. Felegyhazi and T. Holczer, CryPLH: Protecting smart energy systems from targeted attacks with a PLC honeypot, in *Smart Grid Security*, J. Cuellar (Ed.), Springer, Cham, Switzerland, pp. 181–192, 2014.

[2] Y. Huang, A. Cardenas, S. Amin, Z. Lin, H. Tsai and S. Sastry, Understanding the physical and economic consequences of attacks on control systems, *International Journal of Critical Infrastructure Protection*, vol. 2(3), pp. 73–83, 2009.

[3] R. Jaromin, Emulation of Industrial Control Field Device Protocols, M.S. Thesis, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2013.

[4] C. Leita, K. Mermoud and M. Dacier, ScriptGen: An automated script generation tool for Honeyd, *Proceedings of the Twenty-First Annual Computer Security Applications Conference*, pp. 203–214, 2005.

[5] N. Provos, A virtual honeypot framework, *Proceedings of the Thirteenth USENIX Security Symposium*, article no. 1, 2004.

[6] N. Provos and T. Holz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*, Addison-Wesley Professional, Upper Saddle River, New Jersey, 2007.

[7] P. Warner, Automatic Configuration of Programmable Logic Controller Emulators, M.S. Thesis, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2015.

[8] M. Winn, Constructing Cost-Effective and Targetable ICS Honeypots Suited for Production Networks, M.S. Thesis, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, 2015.

[9] M. Winn, M. Rice, S. Dunlap, J. Lopez and B. Mullins, Constructing cost-effective and targetable industrial control system honeypots for production networks, *International Journal of Critical Infrastructure Protection*, vol. 10, pp. 47–58, 2015.