DNIC Architectural Developments for 0-Knowledge Detection of OPC Malware

Julian L. Rrushi, *Member, IEEE*

Abstract—We present an anti-malware solution that is able to reliably detect Object Linking and Embedding for Process Control (OPC) malware on machines in production. Detection is attained on the very first encounter with OPC malware, and hence without any prior knowledge of their code and data. We architected the integration of a decoy network interface controller (DNIC) with a layer of kernel code that emulates a target OPC machine. A DNIC displays a (nonexistent) network, which the compromised machine appears to be connected to. OPC emulation displays a valid (but nonexistent) target OPC machine, which appears to be reachable from the compromised machine over the (nonexistent) network. Our code intercepts OPC malware during their search for target machines over the network. Its overall architecture is crafted to validate the infection by leveraging OPC protocol mechanics. The same principles of operation are used to recognize goodware that access a DNIC by accident. Safe co-existence with production functions and real I/O devices is ensured by a monitor filter driver, which removes all decoy data bound for the monitor. We tested our DNIC architectural developments against numerous OPC malware samples involved in the Dragonfly cyber espionage campaign, and discuss the findings in the paper.

Index Terms—Industrial control systems, malware, defensive deception

1 Introduction

The 0-knowledge detection of industrial control system (ICS) malware provides a formidable defense of the electrical power grid. The ultimate research goal is to detect ICS malware on the first encounter with them, and without any prior knowledge of their code and data. Our line of work explores kernel-level decoys, which unlike honeypots, operate on machines in production and appear to be dynamic. While only a few ICS malware are publicly known to the research community, all of them have been quite effective in operating undetected and hence unhindered by state-of-the-art anti-malware systems for long periods of time. Stuxnet, for example, was discovered after the fact, once it was observed that the centrifuges at a facility were experiencing dysfunction due to anomalies in their rotational speed [1].

Contribution of this Paper. We designed and implemented a cyber deception approach in the kernel of the Windows operating system. Our solution displays, aka projects, a network to which the machine appears to be connected. The displayed network is imaginary, and does not exist. We leverage the Object Linking and Embedding (OLE) for Process Control (OPC) protocol [2] to emulate target OPC machines, and display them as reachable over the imaginary network. The imaginary network is created by a decoy network interface controller (DNIC), which we have integrated with a layer of kernel code that does target OPC emulation. Our work interferes with the target discovery that OPC malware do over the network. OPC malware are intercepted when they pursue a decoy target over the imaginary network.

The approach can co-exist with production functions and real I/O devices because of a monitor filter driver, which removes all decoy data bound for the monitor. The human is kept unaware of the DNIC and emulated ICS machines. The reasoning of this work is specific to the OPC protocol, however it is generally applicable and hence is independent of OPC client/server vendors. No customizations to any specific deployment scenarios are necessary. Our approach runs on machines that are used by human system operators to control and monitor physical equipment and physical processes, such as those of an electrical substation. These machines work in concert with data servers and field devices. Field devices such as programmable logic controllers (PLCs) and intelligent electronic devices (IEDs) are directly attached to physical equipment.

These controllers are the ultimate doers that materialize an action on the physical system. Our approach can also run on any machine in the enterprise network of a power utility company, and indeed on any machine whose owner has offered to run our kernel code. Note that our work is a proof of concept. There are numerous kinds of ICS machines and protocols in operation today, but we had to choose one to make the point, namely demonstrate the security potential of decoy I/O on machines in production. We can use the very same DNIC approach described in this paper to recreate the mechanics of ICS protocols such as S7Comm [3], Distributed Network Protocol (DNP3) [4], IEC 61850 [5]/Manufacturing Message Specification (MMS) [6], and hence detect any ICS malware. We chose the OPC protocol, consequently the ICS targets in this work are emulated OPC servers.

• *The author is with the Department of Computer Science and Engineering, Oakland University, Rochester, MI 48309–4479 USA. E-mail: rrushi@oakland.edu.*

Manuscript received 21 July 2017; revised 24 May 2018; accepted 16 Sept. 2018. Date of publication 28 Sept. 2018; date of current version 15 Jan. 2021. (Corresponding author: Julian L. Rrushi.)
Digital Object Identifier no. 10.1109/TDSC.2018.2872536

Novelty. Our work can attain 0-knowledge detection of OPC malware. Decoy OPC servers are discovered and attacked by OPC malware without a single network packet ever leaving the boundaries of the kernel of the compromised machine. The imaginary network along with the decoy OPC servers are a realistic resemblance of their real counterparts. The architectural developments of this work are crafted to accurately leverage OPC protocol mechanics to detect OPC malware, as well as recognize goodware that access a DNIC by accident. Monitor filtering ensures user safety, which makes our approach usable on a machine in production and reliable in practice.

There are honeypot variants for use in industrial process control networks. One notable example, implemented by Buza et al. is a programmable logic controller (PLC) honeypot known as CryPLH [7]. CryPLH implements several PLC services that are integrated into a Linux-based VM, which forms the honeypot. Another PLC honeypot is CONPOT, which was implemented by Rist et al. [8]. CONPOT relies on Python scripts to emulate a range of common industrial communication protocols. Yet another industrial honeypot was developed by Vollmer and Manic. Their industrial honeypot is able to do self-configuration based on passive observations of control system network traffic [9]

CryPLH, CONPOT, and other industrial honeypots along the same line are characterized by an absence of system and network activity. By contrast, however, ICS machines, such as protective relays in electrical substations, are constantly in action. They read from sensors, analyze data, submit reports to human operators, and send commands to actuators. Industrial process control is a highly dynamic operation full of system and network activities.

Industrial honeypots cannot operate on ICS machines in production, therefore need dedicated computers to run on.

Paper Organization. The remaining of this paper is organized as follows. The threat model is defined in Section 2. In Section 3 we describe the main ideas of our approach, as well as its architecture and main techniques. In Section 4 we describe an evaluation of the performance of our approach in a testbed that resembles an electrical substation. In Section 5 we discuss related work in the areas of anti-malware, cyber deception, and malware's target selection. In Section 6 we discuss improvements and future work. In Section 7 we summarize our findings, and conclude the paper.

2 Threat Model

Our approach can counter a threat model that is representative of all ICS malware that are publicly known to date, namely Dragonfly [10], BlackEnergy [11], [12], Stuxnet [13], and IronGate [14]. Note that some of these ICS malware do not use OPC. Stuxnet, for example, uses S7Comm. Nevertheless, as we wrote previously in this paper, our approach can recreate the mechanics of any ICS protocol, including S7Comm. In addition to exploits, command and control code, and I/O interception and interposition code, ICS malware have code that mimics ICS client software, and hence communicates or attempts to communicate with its targets over the network through an industrial communication protocol. ICS malware also have code that searches for targets prior to pursuing them. The work that we discuss in this paper currently defends from ICS malware that operate in user space. The reader is referred to the section on future work for a discussion of extending this work to counter kernel-space ICS malware as well.

It is noted that some aspects of this work such as self-protection of our code from malware attacks, memory guards, projection of decoy processes and threads, and instrumentation of kernel data structures, are out of the scope of this paper due to room limitations. ICS malware that are known as of this writing penetrate their targets thorough e-mail spear phishing, with attachments that can be malicious Portable Document Format (PDF) files containing embedded JavaScript code, or Microsoft Office files infected through the macro functionality; watering hole attacks injecting an HTML iframe into several websites related to energy, and running the LightsOut exploit kit; and trojanized ICS software that users download from compromised ICS provider websites, and install voluntarily on their machines. Our approach works independently of the exploitation technique used. We wrote previously that our approach is independent of the inner workings of the ICS malware as well.

3 Approach

We now describe the building bricks of this work. The discussion follows a logical order, and exposes both methodical and technical details without loss of generality.

3.1 Core Idea

We postulate that OPC malware that has just landed onto a machine would have to operate like all OPC clients do with regards to accessing the data of OPC servers over the network. They all need to search for, and successfully find, those OPC servers before accessing their data. The search-and-access modus operandi is comprised of the following steps:

- Find all servers that are reachable over the network from the machine on which the code, i.e., OPC client or malware, is running.
- For each server found, query that server for all OPC servers that may be running on it.
- Once a list of OPC servers that are running on a given machine is obtained, use OPC interfaces and identifiers to create OPC objects. Then invoke their methods to read and write data.

Malware Mimics OPC Clients. OPC interfaces and identifiers are discussed in detail later on in this section. At the OPC protocol level, malware has to behave the same as an OPC client. The key difference between the two is that an OPC client will connect to a specific OPC server only if a human operator initiates that connection by operating on the controls of the graphical user interface (GUI) of the OPC client application. Malware, by virtue of being automated attack code, will connect to OPC servers automatically and hence without human intervention. With that said, some malware are driven over a command and control channel, in which a human attacker may enter commands based on the output produced by the malware. However, the human attacker will be using the monitor of his or her own machine when viewing those output data and hence entering commands. As obvious and pointless this observation may

sound at this time, it certainly provides a powerful opportunity for us. Here is why.

In-Kernel Target Emulation with Asymmetrical Views. We emulate a NIC in the kernel of the machine to be defended. The DNIC appears fully functional. There are even decoy processes that appear to be sending and receiving data over the DNIC. In reality they are not. The DNIC projects one or more IP addresses as servers that are reachable over the network from the machine that is to be protected. Both malware and OPC clients can easily find those phantom servers. However, we prevent the monitor of the human operator of an OPC client from displaying the IP addresses of the phantom servers, as well as all data that pertain to the DNIC. The human operator will not see any of the decoy data whatsoever. Given that the human operator is unaware of the DNIC and phantom servers, no connections to those phantom servers will be initiated on the GUI of the OPC client application.

If a connection is made to a phantom server, it must be coming either from malware or from non-OPC applications. These latter may connect to servers automatically or simply send network packets over the DNIC. With the OPC client off the equation, we design and implement emulated OPC servers, and integrate them into the DNIC. The overall architecture of a decoy OPC server perform checks that non-OPC applications cannot pass. If the DNIC receives network traffic that passes those checks, that leaves the malware as the sole source. Our tasks as defenders are much easier on general-purpose machines that are unrelated to industrial process control. The reason is that OPC is a special-purpose protocol for use in industrial control environments, consequently is highly unlikely to be used on machines that have nothing to do with industrial process control. Here any observation of valid OPC traffic flowing over a DNIC is even more indicative of malware.

3.2 DNIC

A Kernel Driver can be Defensively Deceptive. The emulation of a NIC as a decoy I/O device in this work is based on the network driver interface specification (NDIS) virtual miniport driver sample [15], which is provided by Microsoft as part of the Windows 10 driver samples. The NDIS library is implemented by Microsoft as part of the Windows Driver Model (WDM). Most of the functionality of the DNIC lies in a low-level deceptive driver, which is loaded into the kernel and is interwoven with drivers that are specialized in handling real NICs. The driver stack of the DNIC is depicted in Fig. 1. In Windows, the driver stack of an I/O device is an ordered list of device objects, each of which is linked with the driver object of a kernel driver. A device object is a C struct that describes and represents an I/O device [15], whereas a driver object is a C struct that represents the image of a driver [15]. The driver Tcpip.sys sometimes is not considered part of the driver stack of a NIC. Nevertheless, we decided to include it in Fig. 1, given that the low-level deceptive driver interacts with it quite substantially.

The driver stack of a real NIC would typically have one more layer at the very bottom, namely a driver that manages the Peripheral Component Interconnect (PCI) bus. That driver, which is called Pci.sys, was not included in Fig. 1 because the low-level deceptive driver does not communicate with it at all. In fact, the DNIC does not

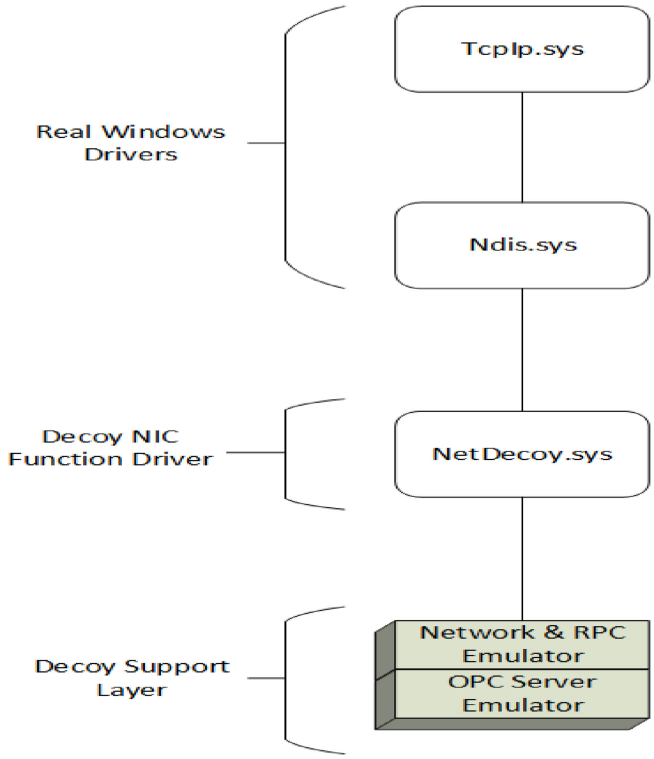

Fig. 1. Decoy NIC.

communicate with the underlying hardware. The network packets that may be sent by malware over the DNIC are packaged into I/O request packets (IRPs) [15]. An IRP is self contained, in the sense that it contains all data that are necessary to describe an I/O request. The Ethernet frames are stored within a member of the IRP structure called SystemBuffer. At the level of the deceptive driver, Ethernet frames can be accessed as a linked list of NDIS data structures called NET_BUFFER. The low-level deceptive driver extracts an Ethernet frame and passes it down to the decoy support layer, instead of placing it on a hardware bus as real NICs do.

The low-level deceptive driver is a miniport driver, which means that by definition it is responsible for handling I/O tasks that are specific to a given NIC, i.e., our DNIC in this case. Those I/O tasks that are common to all network interfaces supported by Windows are handled by the NDIS library. As shown in Fig. 1, the low-level deceptive driver is coupled with the NDIS library, namely Ndis.sys. The low-level deceptive driver serves as a function driver for the DNIC. By definition, the function driver of an I/O device has the most knowledge of how the I/O device operates. The function driver presents the interface of that I/O device to the Windows I/O system.

Factors in Favor. The low-level deceptive driver is favored by two factors, namely its role as a function driver for the decoy I/O device, and the way that the driver stack of an I/O device works in Windows. Consequently, the real Windows drivers that interact with the low-level deceptive driver, as well as the Windows I/O system, base their awareness of the NIC on reports that they receive from the low-level deceptive driver. It is mostly that combination of factors that puts the low-level deceptive driver in the right position to project the existence and operation of a DNIC

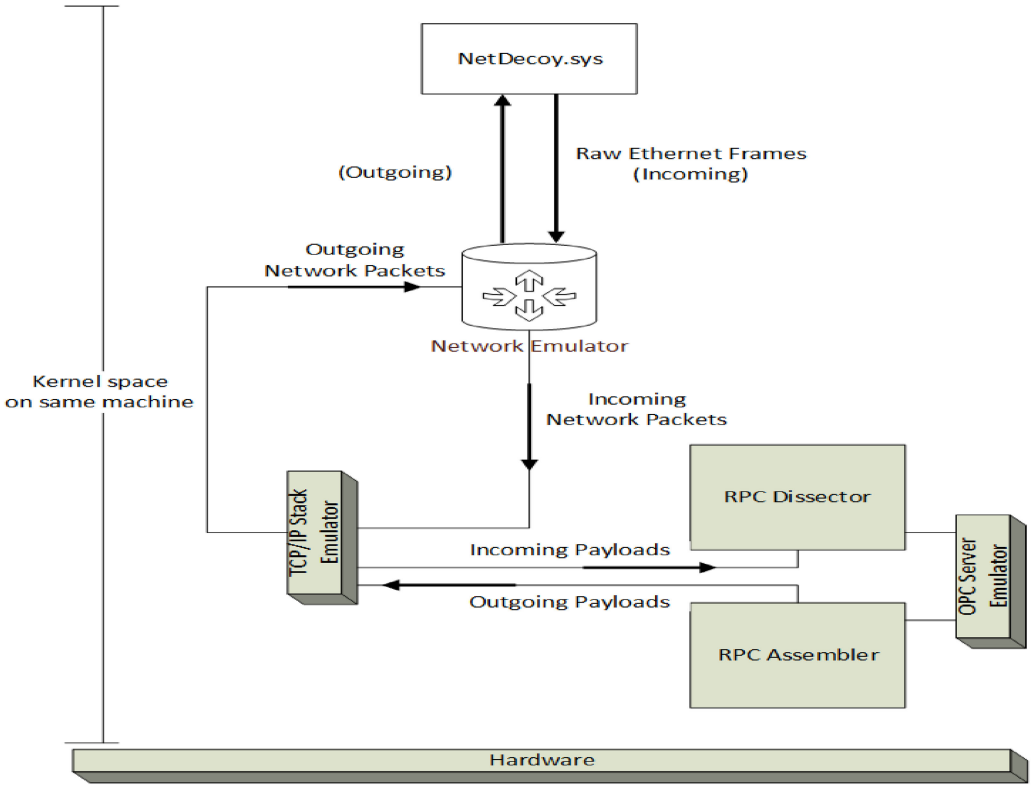

Fig. 2. Network and RPC emulation within the boundaries of the kernel of the same machine.

onto the entire Windows operating system that is running on the machine to be protected.

3.3 Target Emulation

OPC Consists of Objects Communicating over the Network. OPC servers and clients consist of binary code modules that can interact with each other. Those modules are Component Object Model (COM) objects that carry data specific to industrial process control, as well as other data that help with the overall operation of the OPC protocol. OPC classes and hence objects are in fact distributed COM (DCOM), since they are capable of communicating with each other over the network. Except for the network communication, all functions are defined as in COM. The OPC protocol itself is detailed by several specifications. The OPC specification that is the most targeted by attackers is DataAccess (DA), since it allows for reading and writing parameters of industrial processes and physical equipment involved in their monitoring and control. In OPC DA, a server exposes an OPCServer object, which is primarily responsible for maintaining information about the server and, most importantly, serving as a container for OPCGroup objects.

OPCGroup Objects are the Ultimate Target of Attack. There could be several of them provided by an OPC server. OPCGroup objects enable OPC clients to organize process control data that are of interest to a human operator. For example, all parameters of a power transformer in an electrical substation could be organized by a human operator into the same report. Those parameters would be stored as data items within an OPCGroup object. The data items are connections to sources of data, which commonly reside on industrial control systems. Thus, it is the controllers that are directly attached to physical equipment and processes that provide the sources of data for OPC servers to expose to OPC clients, and hence to human operators. In some cases, intermediate industrial control systems, which collect data from controllers that are directly attached to the physical system, could provide sources of data for OPC servers as well.

Associated with each data item is a value, quality, and timestamp. The ultimate goal of malware is to read or write those data items. Reading allows malware to spy on physical processes, as the Havex ICS plugin did. Writing enables malware to directly affect the industrial processes and associated physical equipment, given that data items are simply addresses of data. Writing at an address causes the respective data to change, with the result being that direct physical damage could be initiated on the physical system. The target emulation in this work mimics the behavior and hence responses of an OPC server machine as seen by an OPC client over the network, from the discovery of OPC servers to the actual access to the data items of OPCGroup objects. The architecture of the emulation is depicted in Fig. 2. Remote procedure calls (RPCs) are involved, as DCOM relies on RPC to enable OPC clients to find and access OPC server objects over the network.

Decoy Support Architecture. As it can be seen in Fig. 2, target emulation never extends beyond the boundaries of the kernel of the machine to be defended, although it projects the existence and operation of a network along with OPC server machines. When any code, i.e., malware or goodware, sends network packets over the DNIC, the corresponding IRPs traverse the driver stack of the DNIC. The Ethernet frames eventually reach the function driver, at which point they are ready for transmission if the network was real. Those Ethernet frames are passed down to the OPC decoy support architecture for deep validation.

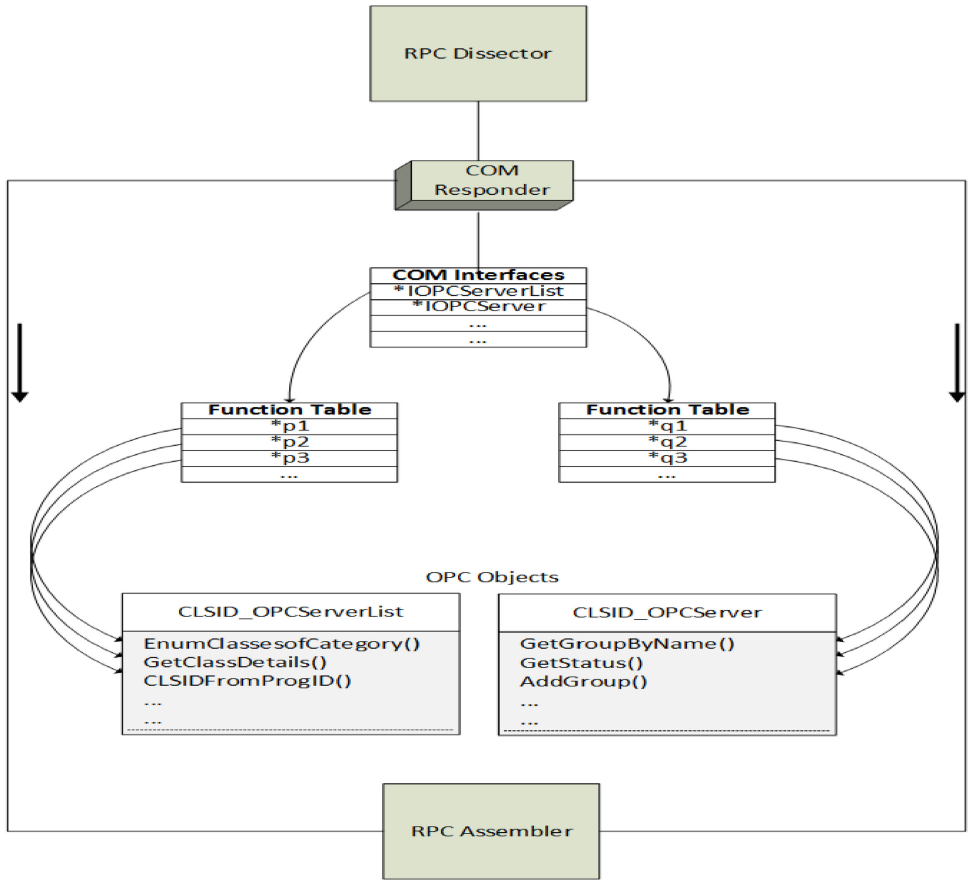

Fig. 3. OPC server emulation.

The Ethernet frames are first received by the network emulator component of our design. The rationale for the network emulator is to reproduce the round-trip network latency of packets. The Ethernet frames are then passed to the TCP/IP stack emulator, which is responsible for emulating the ability of a machine to receive and send network packets. The TCP/IP stack emulator dissects incoming network packets, and also generates reply packets accordingly by formatting them with valid headers and fields of the TCP, UDP, IP, and Ethernet protocols, and possibly with a payload. The TCP/IP stack emulator extracts the payload of an incoming Ethernet frame, which it then sends to an RPC dissector. As of this point, we are not yet in the position to determine whether or not the incoming network traffic was generated by malware.

The RPC dissector extracts the RPC request fragment from the payload, and then extracts the RPC fields. Those fields are sent to the OPC server emulator to convert them into a possible access to OPC objects. If that request for access is valid, in this work we conclude that the caller is malware. In the case of malware detection, the OPC server emulator generates valid OPC output for consumption by malware. That output is marshaled into an RPC message by the RPC assembler component of the OPC decoy support architecture, and thus is passed to the TCP/IP stack emulator. The latter builds a valid network packet with a payload that contains the RPC message, which it then passes to the network emulator. The network packet is delivered to the function driver of the DNIC. The driver stack of the DNIC processes the corresponding IRPs, and delivers the OPC data to the malware.

Validation by Emulation in the Kernel. We now discuss how the OPC decoy support architecture determines whether or not a newly received request for access to OPC objects is valid. More specifically, the validation is performed by the OPC server emulator, which is depicted in Fig. 3. The COM responder component of the OPC server emulator receives the RPC fields from the RPC dissector, and converts them into an actual procedure call. The procedures to possibly invoke are methods of objects, which in turn are instances of C++ classes that mimic those of OPC. In other words, the OPC server emulator resembles the structure and behavior of OPC DCOM classes to provide the same processing of an RPC message as a real OPC server would. The reply to an RPC message is also constructed such as to be consistent with that of a real OPC server. As with a real OPC server, malware is allowed to create OPC objects and invoke their methods.

3.4 Attack Indicators

Correct Navigation Indicates OPC Capability, Good or Bad. We have now reached a critical point in our decision. If the requests that are coming through the DNIC unequivocally issue valid calls to methods of OPC objects, we get the detection mentioned earlier. Those valid calls cannot be coming from non-OPC Windows goodware, including non-OPC DCOM applications. None of those applications can communicate over the OPC protocol, and certainly would never be able to go so far in the interaction with an OPC server, which is of course all emulated in our case. Making it to actually invoking methods of OPC objects involves multiple interactions between the OPC client and server, all

of which have to be formatted correctly and have valid OPC semantics in terms of intended actions. Only malware and OPC clients are able to interact with an OPC server correctly and long enough to progress to the point in which to call methods of OPC objects. And it cannot be OPC clients for reasons that we discuss later on in this section.

In OPC, each DCOM class can be referenced through its Globally Unique Identifier (GUID), which is commonly referred to as class identifier (CLSID). An OPC client, or malware, needs to know the CLSID of an OPC DCOM class in advance before making a reference to that class. The CLSID of each OPC DCOM class is chosen to be unique. As any other GUID on Windows, its being comprised of 128 bits, and hence 16 bytes, makes it highly unlikely for a non-OPC DCOM application or any other Windows goodware to reference an OPC DCOM class by mistake. A CLSID expressed as a sequence of 128 bits is said to be of the REFCLSID data type. A CLSID can also be expressed as a string data type. The string identifier, however, is long enough and unique to exclude the possibility of an OPC DCOM class being referenced as a result of an error.

Another factor that aids a reliable detection of malware is the fact that all calls to OPC object methods are enforced to go through rigidly defined interfaces, and those interfaces can only be referenced unequivocally. An interface in the DCOM and hence OPC context is defined slightly different than in object oriented programming. Here an interface refers to a group of methods implemented by a DCOM class. Those methods may be only a subset of the set of methods that the DCOM class implements. Furthermore, a DCOM class may expose more than one interface, as several OPC classes actually do. At the code level, an interface is a pointer to an array of pointers to methods, i.e., a function table, and those methods are implemented by a DCOM class. As shown in Fig. 3, the COM responder accesses a hash table of COM interfaces. The hash key is the identifier of an interface, while the returned value is a pointer to a function table.

Just like class identifiers, interface identifiers are expressed as a REFCLSID data type, and thus are 128 bit values. They can be expressed as strings as well. The internal referencing of DCOM classes and their interfaces takes place by the means of their identifiers in the REFCLSID form. The string form of those identifiers is mostly meant for use by programmers, and should be converted to the REFCLSID form before being passed to a DCOM and hence OPC server. Similarly to our previous discussion, with 128-bit-long interface identifiers, it is highly unlikely that an application will reference the interface of an OPC class by mistake. Furthermore, each method implemented by an OPC class is invoked by its name. Method names are defined rigidly as well, leaving no room for ambiguity. Clearly the joint probability of a non-OPC application unintentionally referencing an OPC class identifier, an interface identifier, and a method name, all by mistake is now very close to 0.

An Attacker's Entry Gate Consists of Interfaces to OPC Objects. An interface is associated with an OPC object when that object is created. The COM responder is prepared to receive a request from the RPC dissector asking for the creation of an OPC object off a specified OPC class. The COM responder reacts to that request by instantiating the object, and associating the appropriate interfaces with that object just like a real OPC server would. More specifically, the COM responder creates the function table, and populates it with pointers to the methods of the newly created object. Only the methods of the requested interface are covered. The COM responder creates a new entry in the hash table of interfaces, where it stores the pointer to the newly created function table. Since that pointer represents the interface to the newly created object, it is meant for consumption by the OPC client, i.e., malware in our case. The COM responder passes the pointer in question to the RPC assembler, which marshals it into an RPC reply message. That message in turn is processed by the rest of the OPC decoy support architecture, resulting in a network packet that is delivered to the malware.

Now the malware has the interface to the OPC object of interest, namely the pointer to the function table. This is how the protocol flow would have developed if the malware was an OPC client interacting with a real OPC server. In other words, while we are emulating an OPC server to display a target for the malware to hit, we know that the malware has to act like an OPC client to be able to access any OPC server. That is exactly what the Havex ICS plugin had to do. The COM responder now expects to receive from the malware a request that uses the newly obtained interface to call a method of the OPC object. Once that request arrives as output of the RPC dissector, the COM responder checks if the pointer is valid relative to the interface requested. If it is indeed the case that the pointer is valid, the COM responder invokes the requested method of the OPC object referenced in the RPC message.

3.5 Intercepting Malware's OPC Target Discovery

Target Discovery Leads to Detection. The methods implemented by OPC classes mimic the behavior of their real counterparts. Once an invoked method of an OPC object completes and hence returns, the COM responder passes the returned data to the RPC assembler. As before, the RPC assembler marshals those data into an RPC reply message, which is placed in a network packet. That packet in turn undergoes all the processing discussed earlier in this section, and at the end is delivered to the malware. Prolonged interactions with the OPC server emulator depicted in Fig. 3 are not necessary for the detection of malware. The DCOM calls issued by malware to discover all OPC servers on a target machine suffice to detect the malware quite reliably, as we discuss in the experiments section later on in this paper.

Tracking and Counteracting. An OPC server emulator that is capable of sustaining prolonged interaction with malware may be useful if the defender decides to go beyond detection, and thus mounts cyber operations against the attackers. For example, the defender may use the OPC server emulator to provide decoy data about an electrical substation that does not exist. Those decoy data could be designed to be consistent enough to defensively deceive the attackers into engineering a subsequent attack on the phantom electrical substation. Instead of attacking an electrical substation, the malware would end up attacking a honeynet. Active redirection of malware attacks is currently researched by the authors, and will be discussed in future work. Determining what operations the malware is performing on its target (emulated) OPC server is as simple as inspecting the class

identifiers and interface identifiers in the requests sent by the malware to the emulated OPC server.

Leveraging the Server Browser. The discovery of OPC servers that may be running on a target machine is performed by an executable called OpcEnum.exe, which is maintained and hence provided by the OPC Foundation. That executable is referred to as the server browser. It typically runs as a system service, and provides a means to browse the local machine for OPC servers. Each OPC server stores specific configuration data in the Windows registry. The server browser accesses the local Component Categories Manager (CCM), which is a COM object, to get a list of OPC servers, and subsequently sends that list to the OPC client. The data that the server browser sends to the OPC client for each OPC server found on the local machine include a program identifier (ProgID) and a GUID. ProgID is a human friendly name of the OPC server, and typically follows the pattern Manufacturer.ServerName, whereas the GUID is a numerical identifier of the OPC server. Malware obtains exactly the same data as an OPC client would. Those data suffice to connect to an OPC server, and thus read or write data items in its OPCGroup objects.

Malware cannot bypass our OPC decoy support architecture by calling a server browser on another, possibly real, machine, to attempt to discover the OPC servers that may be running on machines, which our DNIC reported as reachable over the network from the compromised machine. By definition, the server browser, i.e., OpcEnum, is only able to browse for OPC servers on the local machine on which it is running. OpcEnum cannot discover OPC servers that may be running on remote machines. OpcEnum implements a DCOM class with CLSID {13486D51-4821-11D2-A494-3CB306C10000}. The string form of that CLSID is CLSID_OPCServerList. The identifier of the interface of that DCOM class is {9DD0B56C-AD9E-43ee-8305-487F3188BF7A}, and has a string form of IID_IOPCServerList2.

Malware detection occurs when the COM responder of Fig. 3 is passed a CLSID equal to {13486D51-4821-11D2-A494-3CB306C10000}, and thus is asked to create an object off the OpcEnum's class. That object is often referred to as OPC Server Browser Object. Typically the same RPC message that requested the creation of the OPC Server Browser Object will also specify an interface identifier such as {9DD0B56C-AD9E-43ee-8305-487F3188BF7A}, asking for a pointer that represents that specific interface. As we wrote earlier in this section, the COM responder creates a mimicked OPC Server Browser Object, and then happily returns the pointer in question. The detection of malware is firmly confirmed when the COM responder receives another RPC message that uses that specific pointer to invoke any of the methods of the OPC Server Browser Object.

3.6 Safety Design

Safety-Friendly Applications. Non-OPC applications do not represent a challenge for this work. As we discussed earlier in this paper, none of those applications have the knowledge and hence code and data to interact with the elements of the OPC decoy support architecture far and long enough to make the COM responder conclude with a malware detection. DCOM applications that are non-OPC are safe too due to several distinct GUIDs involved in the OPC protocol.

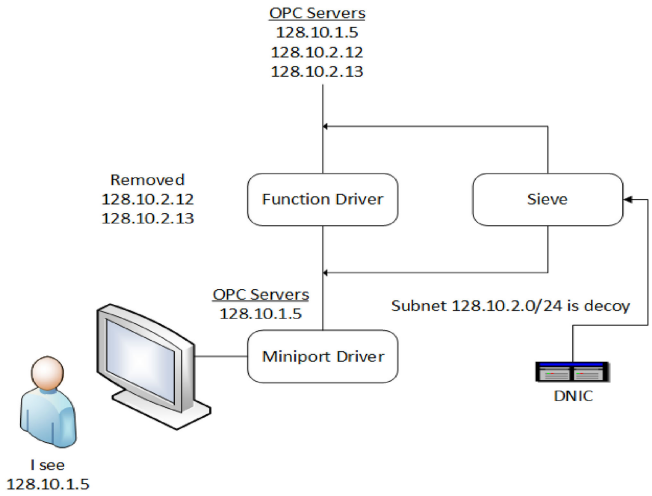

Fig. 4. On-the-fly decoy entry removal.

Challenging Applications. OPC client applications are a challenge, and possibly a deal breaker if not handled well. Those applications are fully capable of discovering and accessing OPC servers, and could very well complete the interaction cycle with our emulated OPC server. An alleviating factor is that OPC client applications are driven by a human operator, who may know or recognize the IP addresses of the OPC server machines that are needed for his or her work. Nevertheless, realistically that may not be always the case, consequently erroneous connections to OPC server emulators are certainly possible.

Filter Driver as a Sieve. We build on our previous work to ensure the usability of our approach. The solution is depicted in Fig. 4, and consists of a kernel driver, namely a filter driver, which is integrated into the driver stack of the monitor. The function driver participates in the preparation of frames of bytes bound for the monitor. This is because of its primary role as a class driver that manages the monitor device. The miniport driver completes its specialized preparation of those frames, which are then displayed on the monitor. The miniport driver is also able to sense and enumerate all monitors that are connected to it. The sieve may attach to the driver stack of the monitor above, or underneath, the function driver, which allows it to intercept the frames of bytes bound for the monitor. The sieve filters out all data that pertain to a DNIC, including the IP addresses of decoy OPC server machines that the DNIC makes appear as reachable from the machine on which the OPC client is running. Those data are filtered out before traveling far enough to be displayed on the monitor. By the time the miniport driver starts its own processing of the data, the IP addresses of decoy OPC servers have been removed.

Marking Data for Removal. The DNIC has code that informs the sieve of the IP address of the decoy network. The defender determines that network address when configuring the DNIC itself, therefore the DNIC simply communicates it to the sieve, as shown in Fig. 4. As a frame of bytes comes through, the sieve simply removes decoy IP addresses, before letting that frame go towards the function driver or miniport driver. The sieve performs a selective inspection of frames of bytes based on the identifier of the process created out of the OPC client executable. The sieve may also take signatures of text to search for and remove.

An example is the output of the `ipconfig` command on a powershell, which includes the IP address of a DNIC. The filtering out of that output will only show the human operator the IP configuration of existent NICs, and will hide the IP configuration of a DNIC.

Effect on the GUI of an OPC Client Application. The filtering done by the sieve only affects the portion of the GUI where IP addresses are displayed. The IP addresses of decoy OPC servers are made blank, whereas the IP addresses of real OPC servers are left intact. As a result of the filtering, a human operator does not see the IP addresses of decoy OPC servers, because they are not visualized on the monitor. Connecting by mistake to decoy OPC servers via the automatic search feature on the GUI of the OPC client application is not possible, as there are no entries about their IP addresses to select or click on. OPC client applications typically provide a panel on the GUI that a human can use to enter the IP address of an OPC server manually and connect to it. If a human operator types the IP address of a decoy OPC server manually, the connection will take place and hence will create a false positive. This scenario may be mitigated by first letting the digits and dots of the IP address be visualized on the monitor one at a time as they come in. As soon as a match with the IP address of a decoy OPC server is found, an error message could be displayed on the GUI to inform the human operator that the connection should not be made.

3.7 IP Addressing

Decoy OPC Servers are Allocated Valid, but Unused, IP Addresses. The imaginary network needs to appear identical to real physical networks in terms of addressing as well. Any minimal inconsistency in IP addressing, as any other inconsistencies of any kind, could inform attackers of the decoy nature of their target OPC servers. The connection of a machine with an imaginary network over a DNIC places an entry in the route table of that machine. Running the `route print` command on the compromised machine, or code that is functionally equivalent to it, shows all entries of the route table, including the route to the imaginary network. Although the human operator does not see this latter route, because we can filter it out before the route table is displayed on the monitor, the machine cannot distinguish it from the other routes. The TCP/IP stack implementation in the kernel will simply use it and hence route packets bound for a decoy OPC server onto the imaginary network, which leads to detection.

At the same time, this also means that there should be no other network segment by the same address, that is reachable by the machine to be protected. Otherwise the two routes would compete with each-other, at times leading attackers to a real machine, at other times leading human operators to a decoy. By keeping the decoy IP address space separate from existing network segments, we enable DNIC not to interfere with IP routing. An illustration is given in Fig. 5. The machine to be protected is connected to an existent subnet and to an imaginary subnet, namely 128.10.1.0 and 128.10.2.0, respectively. Both of these subnets are valid segments of a class B network. Nevertheless, subnet 128.10.2.0 appears to have two OPC servers, in addition to a router that leads to a remote substation network, all of which are decoys.

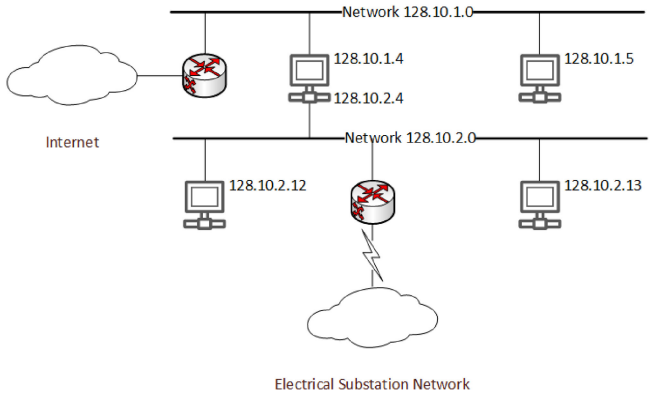

Fig. 5. Example of an imaginary segment of a class B network created via subnet addressing.

Minimizing the IP Address Space Overhead. Small size subnets are common in enterprise networks. These subnets contain internal resources, such as OPC servers in power utility networks, which can be protected with tighter access control policies. We leverage variable-length subnet masks to reduce the size of imaginary networks, if IP address space is a concern. It is typical for system administrators to choose subnet masks on a per-network basis. As few as 2 or 6 machines can be put in a separate subnet, which may be existent or imaginary. Otherwise, many power utility networks have large portions of their IP address space unused, consequently medium-size or long-size imaginary networks are feasible in those cases.

4 Validation

We now discuss the evaluation of this work, as well as the testbed in which the experiments were performed. We used the following metrics to measure the performance of this work:

- The number of distinct ICS malware samples that see the decoy OPC target and pursue it, and any number of ICS malware samples that step away from the decoy OPC target.
- The number of legitimate OPC client applications that somehow access a DNIC by accident during their operation.
- The number of legitimate applications, OPC or non-OPC, that access a DNIC by accident and are safely ignored by our approach, or any number of them that cause our approach to erroneously conclude that there is an attack in progress.

4.1 Testbed

Real-World ICS Machines. We created a computer environment that is similar to that of a real-world electrical substation. The ICS machines in the testbed were all real, and are shown in Fig. 6. ICS machines of that kind are widely deployed in electrical substations in North America. Their main characteristics are summarized in Table 1. More specifically, the SEL-487E-3 is an ICS that can monitor and protect a power transformer from electrical faults. It runs intelligent algorithms to detect various types of faults, and can take action in a timely manner by operating electrical circuit breakers and disconnect switches. The SEL-421-4 can

Fig. 6. Some of the machines in our substation testbed.

TABLE 1
Testbed Machines and their Main ICS Protocols

Machine	ICS function	ICS protocol
SEL-3555	Automation controller	IEC 61850
SEL-487E-3	Transformer protection relay	IEC 61850
SEL-421-4	Protection, automation, and control	IEC 61850
General-purpose Windows server	OPC server	OPC, IEC 61850
General-purpose Windows client	OPC client	OPC

perform industrial automation functions. It includes 32 programmable elements for local control, remote control, automation latching, and protection latching. A SEL-421-4 can also conduct various functions to protect overhead electrical transmission lines and underground cables. The SEL-3355 conducts multiple substation functions too. It has an integrated human-machine interface (HMI), with a local display port. In this work, the SEL-3355 was used as a real-time automation controller. The SEL-3355 polls the SEL-487E-3 and SEL-421-4 to collect substation data from them.

Substation Testbed uses OPC, IEC 61850, and MMS Protocols. The network communications between these ICS machines take place over the IEC 61850 protocol. IEC 61850 is a virtual protocol and thus cannot be used alone for concrete network communications. It needs an actual carrier protocol. Consequently, IEC 61850 is typically mapped to a protocol stack comprised of MMS, TCP/IP, and Ethernet. MMS was designed specifically for transferring in real time physical process data and supervisory commands. MMS has object models and services that can easily accommodate those of the IEC 61850 standard. IEC 61850 consists of abstract models of data and services. One of the fundamental models of IEC 61850 is that of logical node. A logical node is a virtual representation of a real piece of equipment or function. Logical nodes are grouped according to the function that they carry out in the electrical substation.

For example, directional comparison (PDIR), distance protection (PDIS), and directional overpower (PDOP), are all logical nodes in the protection functions group. Similarly, virtual circuit breakers (XCBR) and virtual circuit switches (XSWI) are logical nodes in the switchgear group. A logical device is a composition of logical nodes that have common features. A physical device represents the IED where logical devices reside. Network communication services in IEC 61850 are also abstract and model-driven. Those services are defined by the Abstract Communication Service Interface (ACSI), and are organized in two groups. In one group, the services follow a client-server model. Those services typically set or get data values. In the other group, the services follow a peer-to-peer model, in which data are sent fast and reliably from one IED to a group of other IEDs. The data exchanged between IEDs over the network are referred to as Pieces of Information for Communication (PICOM).

The path established for transmission of PICOMs over the network is referred to as an association. The mapping of IEC 61850 data and services to concrete network

communication protocols is done by the Specific Communication Service Mapping (SCSM). SCSM defines the syntax and encoding of messages, as well as how those messages are passed over the network.

Substation Testbed Includes OPC Server. We integrated an engineering server into the testbed. The engineering server hosted an OPC server, which in turn ran an IEC 61850 protocol driver to get substation data from the ICS machines. Various attributes of logical nodes on the ICS machines were marshaled into PICOMs, and hence were sent by ACSI services to the OPC server over the IEC 61850 protocol using a simple association. There, those data were received by the IEC 61850 protocol driver, and were subsequently stored in the OPC server as data items of OPCGroup objects. We added an OPC client machine to the testbed. The OPC client application has a graphical user interface for a human operator to enter commands using a keyboard and mouse. The OPC client machine runs a DNIC supported by a decoy layer in its kernel. All machines in the testbed were connected on a local area network.

4.2 Experimentation

First of all, Safety. We ran many OPC client applications on the OPC client machine, namely MatrikonOPC explorer, Kepware ClientAce, Kassl dOPC Explorer, PowerOPC DA client, OPCconnect test client, Integration Objects OPC DA test client, CommServer OPC Viewer, and the Prosys OPC Client. Several servers appeared to be reachable over the DNIC as a result of the defensive deception operated by our approach. As we ran each OPC client, we tried to use its graphical user interface to select the IP address of any of the decoy servers, and thus connect to it. We ran the experiment of connecting to the decoy servers under the assumption that we did not know their IP addresses, unless those IP addresses were displayed on the graphical user interfaces of the OPC clients. The OPC clients did not show any IP addresses that belong to the decoy OPC servers.

There was no IP address entry to click on. The `Connect` button remains disabled if no valid IP address entry is selected. The combination of those factors prevented us from connecting to any of the decoy OPC servers. Making a connection to a real OPC server requires the human operator to enter the IP address and name of the OPC server. The same procedure could be used to connect to a decoy OPC server. However, the human operator never sees the IP addresses and names of decoy OPC servers. Without any knowledge of those data, the procedure does not initiate the connection. The only way that decoy OPC servers could be discovered is by starting the search procedure on the graphical user interface of the OPC client. That search procedure does find the decoy OPC servers, however those are not displayed, unlike the real OPC server entries, which are retained and hence seen by the human operator.

We ran a large number of goodware on the OPC client, and also started all services on the machine. The DNIC did receive network traffic, however none of that traffic passed the deep checks performed by the decoy support layer. The network packets never made it to instantiate the OpcEnum class, let alone invoke its methods. No malware detection was reported.

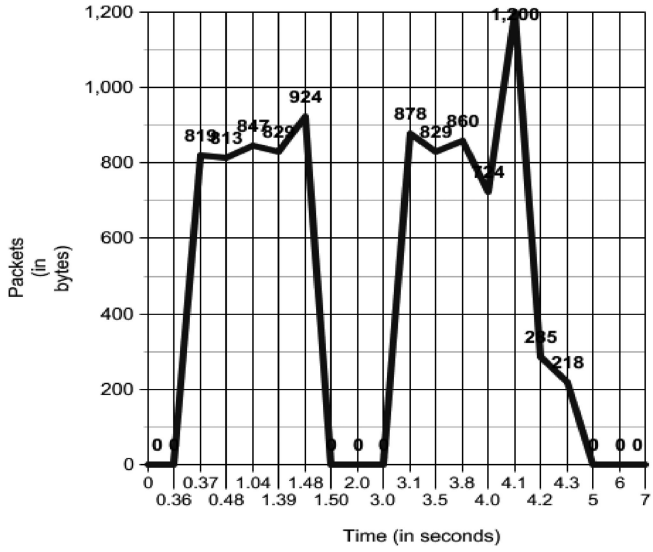

Fig. 7. Ethernet frame arrivals on a DNIC over time.

Now Effectiveness. We were able to perform a qualitative validation of the approach, rather than a quantitative validation. The reason is that not many ICS malware have been detected as of this writing. There is no large ICS malware sample set that we could use for testing purposes in this work. Of those ICS malware that are available for research analysis, only a few are based on the OPC protocol. We decided to validate this work against the OPC malware involved in the Dragonfly cyber espionage campaign. A sense of quantity is provided by the fact that there are so many versions of those malware. We obtained numerous OPC malware samples from public research malware repositories and thus used those versions in our experiments. The various versions of the Havex ICS plugin are of particular interest to this work, since they all use OPC. In the rest of this section, we discuss our findings as applied to those samples.

We ran the malware samples on the OPC client machine one at a time, and observed whether or not our approach would report the detection of each sample. As an aside, we conducted a round of trials in which we laid aside our approach and allowed the ICS malware samples discover a real target in the substation testbed. We did this to validate their effectiveness as well, as we did not want to test our work against ineffective malware. All malware samples that we ran managed to discover the real target, namely the engineering server, connected to its OPC server, and collected over OPC the electrical substation data originating in the ICS machines over IEC 61850. Once our approach became active again, the malware samples were intercepted by it when they attempted to create an OPC Server Browser Object. Some of the Ethernet frame arrivals, as observed in the DNIC, are graphed in Fig. 7.

A message is sent over the imaginary network through the DNIC, and thus traverses all components of the decoy support layer depicted in Fig. 2. It enters the OPC server emulator, and hence reaches the COM responder. This latter detects the reference to the DCOM class with identifier CLSID_OPCServerList, and thus creates the OPC Server Browser Object. The COM responder detects also the reference to the identifier of the interface of that DCOM class,

namely IID_IOPCServerList2. We are writing here the string form of those identifiers to help with the readability of this section. In reality, the COM responder encounters the class identifier and the interface identifier in their REFCLSID form, and hence as a sequence of 128 bits. Here is a snippet of code where a malware sample makes the request, which causes the generation of the message in question. At point (1), `Clsid` is the REFCLSID form of the DCOM class identifier CLSID_OpcServerList. At point (3), we see `pResults`, which is a pointer to an array of the so called MULTI_QI structures. There is only one MULTI_QI structure in that array, as indicated by register `%ebx` at point (2). Immediately prior to point (2), we can see register `%ebx` getting zeroed out by the xor instruction, and then incremented by 1.

```
lea     eax, [esp+8Ch+pResults]
push    eax                                     (3)
xor     ebx, ebx
inc     ebx
push    ebx                                     (2)
lea     eax, [esp+94h+pServerInfo]
push    eax
push    17h
push    edi
push    offset Clsid                            (1)
mov     [esp+0A4h+var_4], edi
call    ds:CoCreateInstanceEx
```

A MULTI_QI structure, as used by these malware samples, is a placeholder for the pointer that represents the interface IID_IOPCServerList2 of the DCOM object CLSID_OpcServerList. At the moment in which the call to `CoCreateInstanceEx()` is issued, that pointer is NULL, since the request has not been sent yet. Once the message is processed by our approach, the COM responder returns a valid value for that pointer, making it point to the function table shown on the left part of Fig. 3. There is another member of the MULTI_QI structure that is of great significance to the COM responder. That member is a pointer to the identifier of the interface that the caller, i.e., malware samples in this case, intends to request from the target OPC server. That member is the origin of the interface identifier IID_IOPCServerList2 in its REFCLSID form that is detected by the COM responder.

Here is another snippet of code where a malware sample prepares the specific MULTI_QI structure that led to detection by the COM responder. The pointer to the interface identifier IID_IOPCServerList2 in its REFCLSID form is used at point (3). We show those 128 bits as dumped from the memory of a malware sample later on in this section. The pointer that represents the interface IID_IOPCServerList2 of the DCOM object CLSID_OpcServerList is used at at point (2). That pointer is initialized to NULL on input as per our previous discussion. At point (1), we see a status field that will indicate whether or not the interface was found and obtained successfully. The decoy support layer overall operated consistently, and thus caused the return of a value of `S_OK` for that status field, which means that the pointer that represents the requested interface was obtained successfully. `S_OK` has a concrete binary value of 0.

```
mov     eax, [esp+98h+var_78]
add     esp, 0Ch
push    edi
push    edi
mov     [esp+94h+pServerInfo.pwszName], eax
mov     [esp+94h+pResults.pIID], offset         (3)
        unk_10030C78
mov     [esp+94h+pResults.pItf], edi            (2)
mov     [esp+94h+pResults.hr], edi              (1)
```

The detection by the COM responder of a call to a method of the DCOM object CLSID_OpcServerList is the last step needed by the decoy support layer to conclude the detection of malware. Here is a snippet of code where a malware sample issues one of those calls. The code first checks at point (4) if the pointer returned by the target OPC server, i.e., the decoy support layer in this case, is NULL. If that is the case, at point (3) the code performs a jump and thus does not proceed with the invocation. The decoy support layer operated correctly, consequently the pointer in question was valid. The code proceeded with the invocation. At point (2) the code retrieves from memory the pointer to the beginning of the function table shown on the left part of Fig. 3, and subsequently uses it at point (1) to issue the method invocation.

```
cmp             ecx, edi                        (4)
jz              short loc_10001C79              (3)
mov             eax, [ecx]                      (2)
lea             edx, [esp+8Ch+var_64]
push            edx
push            edi
push            edi
push            esi
push            1
push            ecx
mov             ebx, ecx
call            dword ptr [eax+0Ch]             (1)
```

Here is a memory dump of the 128 bits of the CLSID of the DCOM class CLSID_OPCServerList, which were detected when creating an OPC Server Browser Object on the decoy support layer.

```
.rdata:10030C68 Clsid dd 13486D51h
.rdata:10030C68 dw 4821h
.rdata:10030C68 dw 11D2h
.rdata:10030C68 db 0A4h, 94h, 3Ch, 0B3h, 6, 0C1h, 2
dup(0)
```

In conclusion, here is a memory dump of the interface identifier IID_IOPCServerList2 from the memory of a malware sample. These bytes were detected when requesting the pointer to the function table shown on the left part of Fig. 3.

```
.rdata:10030C78 unk_10030C78    db      6Ch
.rdata:10030C79                 db      0B5h
.rdata:10030C7A                 db      0D0h
.rdata:10030C7B                 db      9Dh
.rdata:10030C7C                 db      9Eh
.rdata:10030C7D                 db      0ADh
.rdata:10030C7E                 db      0EEh
.rdata:10030C7F                 db      43h
.rdata:10030C80                 db      83h
```

```
.rdata:10030C81                db      5
.rdata:10030C82                db      48h
.rdata:10030C83                db      7Fh
.rdata:10030C84                db      31h
.rdata:10030C85                db      88h
.rdata:10030C86                db      0BFh
.rdata:10030C87                db      7Ah
```

Access to the OPCServer object shown on the right part of Fig. 3, as well as access to the various OPCGroup objects, follow similar routes of processing within the decoy support layer. They are detected by the COM responder using the same techniques discussed in this section and earlier in the paper.

We performed further testing to replay the BlackEnergy style of attacks as well. BlackEnergy used a keylogger module to intercept keystrokes. It intercepted VPN credentials on compromised machines in the enterprise network of a target utility company, and subsequently used those credentials to access substation networks over a VPN. All this enabled attackers to run rogue SCADA client software on the compromised machines similarly to how legitimate operators do [12]. We involved in the tests a decoy keyboard, which we developed in our other work discussed in [16]. The decoy keyboard uses a low-level driver to emulate and expose keystrokes modeled after actual users. The decoy keyboard is able to shadow the physical keyboard, such as one single keyboard appears on the device tree of the operating system at all times. That keyboard is the physical keyboard when the actual user types on it, and the decoy keyboard during time windows of user inactivity.

The decoy keyboard leaked decoy credentials of a VPN, which appeared reachable only over the DNIC and led to an imaginary substation network. Those decoy credentials were picked up, and were subsequently used in conjunction with SCADA client software to read and write the data of a decoy OPC server situated in the imaginary substation network. The attack was intercepted by our approach shorty after it started, with OPC mechanics and data arrivals on the DNIC that are similar to those discussed previously in this section.

Efficiency. In this work, we express overhead as the approximate average fraction of 1 millisecond that goes to executing instructions of deception code. Although this measure is a statistical estimate, it clearly quantifies the effort that the CPU makes to run our approach. When the machine is not under attack, the execution overhead is absent or at most double-digit picoseconds, in which case it is due to safety actions and ICS consistency measures done in the kernel. When under attack, the execution overhead increases to under two nanoseconds due to full ICS protocol emulation and caller localization. A sample of overhead measurements is given in Fig. 8.

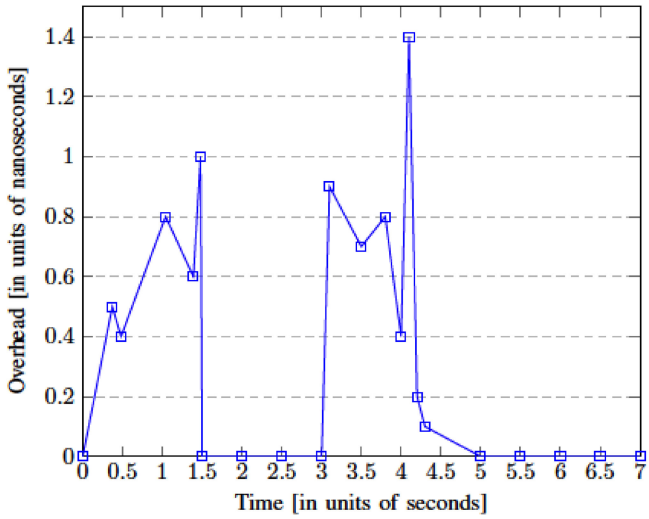

Fig. 8. Execution overhead over time.

5 Related Work

5.1 Malware Analysis

Various malware detection approaches rely on analyzing the code and data structures of executable files. Furthermore, current malware detection tools require that an instance of malware be given to them in input for analysis. That malware sample needs to be detected by some other means, which are often honeypots. Honeypots have their own limitations. Read next section for a related work discussion of current decoy tools and techniques, including honeypots. Someone has to provide the sample of malicious code, along with an explicit and validated finding that the sample is indeed malware. Those tools analyze the sample, and thus learn various signatures and indicators, which can subsequently be used to detect other instances and variants of the malware. The code analysis approaches can be categorized into static and dynamic. Static analysis techniques work by scanning an executable file for specific strings or instruction sequences that are unique to specific malware samples [17].

One of the drawbacks of static analysis techniques is that malware can change its appearance by means of polymorphism, metamorphism, and code obfuscation. Code mutation is often effective in invalidating malware signatures that were preliminarily learned from one or more samples. Several static analysis works focused on higher-order properties of executable files, such as the distribution of character n-grams [18], control flow graphs [19], and function recognition [20]. Dynamic analysis techniques detect malware code by analyzing its execution, typically on virtual machines but at times on physical machines as well. Those techniques try to learn various malware indicators, which are used later in the actual detection phase. Detection indicators include disk access patterns [21], malspecs [22], sequences of system calls and system call parameters [23], and behavior graphs [24].

The main challenge in dynamic code analysis is the use of anti-debugging and anti-virtualization techniques [25], [26] by malware, as well as stalling code [27]. We discuss anti-debugging and anti-virtualization later on in this section. The stalling code technique consists of delaying the execution of malicious operations for so long that dynamic analysis tools conclude that the executable is not malicious, given that the malicious operations were not observed. Another malware detection approach examines the data structures in an executable file [28]. The assumption is that malicious code will morph into different variants, however its data structures will remain the same and thus can be used to detect the malware. Nevertheless, data structure layout randomization changed the rules of the game [29].

Randomizing data structures is more difficult than changing the code, however it is certainly doable. Consequently, malware can change its data structures as well as its code to evade detection.

The DNIC approach overcomes the limitations of the current body of malware detection tools and techniques in that it does not require or need any prior analysis of a malware sample for being able to detect that malware. Malware is detected on the first encounter with it solely on the basis of its interaction with a DNIC. Furthermore, the DNIC approach invalidates all the counter anti-malware techniques discussed previously in this section. None of those techniques would enable malware to evade detection by a DNIC.

5.2 Current Decoy Tools and Techniques

Most of the current cyber deception capabilities can be categorized into decoy data and honeypots. Honeyfiles for example are a form of decoy data. Honeyfiles are bait files that reside on a computer and are intended for attackers to access [30], [31]. A honey-file system was developed to monitor all accesses to honeyfiles on a computer. When an access to a honeyfile takes place, the honey-file system raises an alert and also adds an entry to a log file to record the access. Honeyfiles were originally intended to be stored on a honeypot, but were later adapted for deployment on computers in production [32]. Another form of decoy data are decoy database records [33]. Decoy records are inserted into one or more tables on a database server in production, and the accesses to them are monitored. Similarly to a honeyfile, any access to a decoy record on the database server is leveraged as an indication of malicious activity, which results in an alert being raised.

Decoy software is a variation of honeyfiles, and is discussed in [34]. The proposed approach generates Java source code that deceptively appears as valuable proprietary software. The deception target here is an attacker who aims at stealing software that implements intellectual property algorithms. Various beacons are inserted into the deceptive software such to connect to a server operated by the defender, and thus report the time and location of the machine the software was run on. Beacons are implemented by inserting crafted Javascript code into PDF and HTML files. Those files are made to appear as the documentation that accompanies the proprietary software, which is in fact a decoy. When those files are viewed by the attacker, the Javascript code runs as well and hence calls and reports home. Research works such as the Decoy Document Distributor (D3) System [35], and the Decoy Distributor Tool (DDT) [36], automated the generation, distribution, and monitoring of decoy documents. The writing of decoy documents in a foreign language was explored as a means of helping the detection of the attack. The attacker would need to exfiltrate the contents of those documents in order to translate them [37].

The main limitation of decoy data and code lies in their not being accessed by goodware applications. Malware could simply observe all I/O traffic to and from any files on the compromised machine, including honeyfiles, honeycode, decoy documents, and database files. If the malware observes an absolute absence of I/O traffic to and from a target file for a long time, the conclusion could be that the file is likely to be a decoy. The very same strength of honeyfiles, namely the fact that any access to those files denotes malicious activity, now becomes their main weakness. Traditional honeypot solutions such as Honeyd [38], BitSaucer [39], Shark [40], Sebek [41], etc., have a similar limitation. They all have no system or network activity until they are attacked, which makes them relatively easy to detect.

High interaction honeypots [42] cannot be mixed with production functions, because the legitimate system and network activity that those functions produce may be interpreted as malicious. It is for this reason that high interaction honeypots are not deployed on machines in production, and hence typically occupy a separate machine, which may be physical or virtual. Honeypots also lack the human factor. Since there is no human interaction on a honeypot, the enablers of the Havex ICS plugin such as email spear phishing campaigns and watering hole attacks are missed. Honeyclients perform better, since they can emulate human interaction with a website [43]. Nevertheless, honeyclients do not work against trojanized versions of legitimate software packages, since no browser attacks take place. The trojanized software is simply installed by the user, who is not aware of the malicious code and hence trusts the source of the software. Furthermore, honeyclients are susceptible to a myriad of evasion techniques, which allow attackers to mount a malware attack without being detected. Some of those techniques are discussed in [44], [45].

The DNIC approach overcomes all of those limitations. A DNIC appears to be in operation, and hence is dynamic to maintain consistent resemblance to its real counterparts over time. A decoy process in user space appears to send and receive traffic that appears to be transported by the DNIC. The process identifiers of the decoy process vary from time to time in order to mimic the creation and termination of processes that communicate over a network. The frequency of decoy process creation as well as the rate at which the DNIC appears to send or receive traffic on behalf of the decoy processes varies according to a pseudorandom function. The DNIC can safely coexist with the software and hardware functions of a machine in production, where the human factors are all real. Trojanized versions of legitimate software packages are detected based on their interaction with a DNIC, just like any other malicious code that makes use of the network.

5.3 Malware Target Selection

Various aspects of the Havex malware, and most importantly of its ICS plugin, have been reverse engineered and analyzed by malware researchers with FireEye and F-Secure [46], [47]. In this work we are concerned with the target selection process implemented in the Havex ICS plugin. Malware in general-purpose computing are known to perform target selection based on various forms of validations. One of those validations pertains to verifying that a compromised machine is not simply a trap which is aiming at aiding a forensics analyst to uncover the operations of the malware [25]. Most dynamic code analysis projects are commonly implemented through the use of a debugger tool, and often on a virtual machine. Advanced malicious codes actively use various creative anti-debug techniques to detect the presence of a debugger on a running machine.

Once a debugger is detected, the malware take action to hinder code analysis. Some of those actions create frustration for the analyst, while others cause a significant time delay into the overall code analysis process. Other more deceptive actions include:

- Skipping the execution of blocks of code that are otherwise critical to the understanding of what the malware does for real.
- Executing blocks of malicious code with data that conceal their actual behavior.
- Executing dummy blocks of code that are there only to perform dummy computations to appear as if the malware is doing something useful.

All those actions can deceive the forensic analyst. Similarly, malware can utilize a variety of techniques to discover whether or not the machine on which it is running is virtual rather than physical [25]. Other validations that are typically performed by malware pertain to detection of central processing unit (CPU) emulators [26]. The target selection process mounted by the malware is directly affected by those findings. If the malware discovers that some or all of its underlying execution environment is a decoy, it does not proceed with pursuing its targets.

Stuxnet is known to perform some validation when searching for its WinCC targets. The attack code actively searches for Step 7 project files on the filesystem of a compromised machine. The attack code, however, excludes Step 7 project files that are found under path names that match `*n`Step7`n`Examples`n`*. Those are merely sample project files, which serve as a model for developers to follow when developing their real projects. The conjecture made by the Stuxnet developers is that sample Step 7 project files are unlikely to be used in production, and consequently are not viable targets for infection. The attack code also searches for `.mcp` files, which trigger Step 7 project infection and WinCC database infection. The attack code also monitors the data blocks of a PLC, with the purpose of infecting specific types of Simatic PLCs [13].

6 Future Work

The network emulator shown in Fig. 2 currently produces round-trip latencies for network packets based on empirical measurements, which we made on the local area network in our lab beforehand. Clearly that is a limitation, since the round trip latencies of a single network would be displayed on all locations where the DNIC in conjunction with the decoy support layer would operate. This part of the project is work in progress, which we plan to complete within the next few months. We are exploring machine learning techniques to model and analyze the network delays of a network with an arbitrary number of nodes and topologies. We have reviewed related works on network modeling, which we deem may have some applicability to our research problem. With that said, we are seeking a practical approach, which we could add to the code of the network emulator and run in practice.

Our research project aims at extending the work that we discussed in this paper such as to counter kernel-level malware as well. In that case, however, we need to discover techniques to enable our approach to defend its own code and data from the malware, and also do memory monitoring and tracking in the kernel. Clearly kernel-level malware may decide to access the hardware bus directly, and thus query a target NIC without going through its driver stack. We are exploring a solution based on hardware support, which will mimic NIC replies on the hardware bus such as to pass all tests posed by probes mounted by malware.

Another specification has been added to the OPC protocol, namely the OPC Unified Architecture (UA). OPC UA does not depend solely on Windows. It has a cross-platform and service-oriented architecture that is suitable for process control. Our work can easily be extended to cover OPC UA, as well as any other ICS protocol. Same principles of operation apply. Servers on Windows can be alternatively discovered over the network through a computer browser or the remote registry. The same filtering and detection techniques can be used in those cases as well.

7 Conclusions

This work showed the potential of in-kernel OPC protocol emulators and DNICs to detect OPC malware. The key intervention that disrupts OPC malware is the display of DNICs that are dynamic and consistent enough to resemble their real counterparts, as well as the incorporation of an emulated OPC protocol server underneath the driver stack of each DNIC. The filtering of decoy data from the monitor of a human operator provides for a formidable usability of this approach. The human operator is protected by not seeing entries that pertain to the decoys, consequently any interaction by mistake with the decoys on his or her part is prevented proactively. Numerous OPC malware samples involved in the Dragonfly cyber espionage campaign were detected by our approach quite reliably, and without any false negatives. This approach is highly immune to goodware, including those that use libraries such as DCOM. In conclusion, this approach can obtain malware detection without any false positives, and most importantly without any prior knowledge of the OPC malware.

Acknowledgments

This research is sponsored by the Air Force Office of Scientific Research and the U.S. Air Force Academy Center for Cyberspace Research under agreement number FA7000-16-2-0002. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force, Department of Defense, or the U.S. Government.

References

[1] K. Zetter, *Countdown to Zero Day: Stuxnet and the Launch of the World's First Digital Weapon Hardcover*, 1st ed. New York, NY, USA: Crown, Nov. 2014.

[2] J. Lange, F. Iwanitz, and T. Burke, *OPC - From Data Access to Unified Architecture*, 4th ed. Berlin, Germany: VDE Verlag GmbH, 2010.

[3] Siemens, "What properties, advantages and special features does the S7 protocol offer?." [Online]. Available: https://support.industry.siemens.com/cs/document/26483647/what-properties-advantages-and-special-features-does-the-s7-protocol-offer-?dti=0&lc=en-WW

[4] DNP Technical Committee, "Distributed network protocol." [Online]. Available: https://www.dnp.org
[5] International Electrotechnical Commission, "IEC 61850: Communication Networks and Systems in Substations," *Int. Standard*, parts 1 through 9, 1st ed. 2004.
[6] International Organization for Standardization, Technical Committee 184, "Manufacturing message specification." [Online]. Available: https://www.iso.org
[7] D. I. Buza, F. Juhasz, G. Miru, M. Felegyhazi, and T. Holczer, "CryPLH: Protecting smart energy systems from targeted attacks with a PLC honeypot," *Smart Grid Secur.*, vol. 8448, pp. 181–192, Aug. 2014.
[8] L. Rist, J. Vestergaard, D. Haslinger, A. De Pasquale, and J. Smith, "CONPOT ICS/SCADA honeypot." [Online]. Available: http://conpot.org
[9] T. Vollmer and M. Manic, "Cyber-physical system security with deceptive virtual hosts for industrial control networks," *IEEE Trans. Ind. Informat.*, vol. 10, no. 2, pp. 1337–1347, May 2014.
[10] Symantec, "Dragonfly: Cyberespionage attacks against energy suppliers," Jul. 2014. [Online]. Available: https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/Dragonfly_Threat_Against_Western_Energy_Suppliers.pdf
[11] ICS-CERT, "Cyber-attack against ukrainian critical infrastructure." [Online]. Available: https://ics-cert.us-cert.gov/alerts/IR-ALERT-H-16-056-01
[12] R. M. Lee, M. J. Assante, and T. Conway, "Analysis of the cyber attack on the Ukrainian power grid," defense use case white paper, Mar. 2016. [Online]. Available: https://ics.sans.org/media/E-ISAC_SANS_Ukraine_DUC_5.pdf
[13] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet dossier," Symantec Security Response, version 1.4, Feb. 2011. [Online]. Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf
[14] J. Homan, S. McBride, and R. Caldwell, "IronGate ICS malware: Nothing to see here... Masking malicious activity on SCADA systems," FireEye threat research blog, Jun. 2016. [Online]. Available: https://www.fireeye.com/blog/threat-research/2016/06/irongate_ics_malware.html
[15] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Parts 1 and 2*, 6th ed. Redmond, WA, USA: Microsoft Press, 2012.
[16] S. Simms, M. Maxwell, S. Johnson, and J. Rrushi, "Keylogger detection using a decoy keyboard," in *Proc. IFIP Annu. Conf. Data Appl. Secur. Privacy*, Jul. 2017, 433–452.
[17] P. Szor, *The Art of Computer Virus Research and Defense*. Reading, MA, USA: Addison Wesley, Feb. 2005.
[18] W. Li, K. Wang, S. Stolfo, and B. Herzog, "Fileprints: Identifying file types by n-gram analysis," in *Proc. 6th Annu. IEEE SMC Inf. Assurance Workshop*, Jun. 2005, pp. 64–71.
[19] L. Martignoni, D. Bruschi, and M. Monga, "Detecting self-mutating malware using control flow graph matching," in *Proc. Conf. Detection Intrusions Malware Vulnerability Assessment*, Jul. 2006, pp. 129–143.
[20] M. Christodorescu, S. Jha, S. A. Seshiam, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proc. IEEE Symp. Secur. Privacy*, 2005, pp. 32–46.
[21] A. Felt, N. Paul, D. Evans, and S. Gurumurthi, "Disk level malware detection," in *Proc. Poster: 15th Usenix Secur. Symp.*, 2006.
[22] M. Christodorescu, C. Kruegel, and S. Jha, "Mining specifications of malicious behavior," in *Proc. 6th Joint Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 5–14.
[23] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, "A quantitative study of accuracy in system call-based malware detection," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 122–132.
[24] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, "Effective and efficient malware detection at the end host," in *Proc. 18th Conf. USENIX Secur. Symp.*, 2009, pp. 351–366.
[25] X. Chen, J. Anderson, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proc. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, Jun. 2008, pp. 177–186.
[26] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Proc. 10th Int. Secur. Conf.*, Oct. 2007, pp. 1–18.
[27] C. Kolbitsch, E. Kirda, and C. Kruegel, "The power of procrastination: Detection and mitigation of execution-stalling malicious code," in *Proc. 18th ACM Conf. Comput. Commun. Secur.*, Oct. 2011, pp. 285–296.
[28] A. Cozzie, F. Stratton, H. Xue, S. T. King, "Digging for data structures," in *Proc. 8th USENIX Symp. Operating Syst. Des. Implementation*, Dec. 2008, pp. 255–266.
[29] Z. Lin, R. D. Riley, and D. Xu, "Polymorphing software by randomizing data structure layout," in *Proc. 6th Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, Jul. 2009, pp. 107–126.
[30] C. Stoll, *The Cuckoo's Egg*. New York, NY, USA: Doubleday, 1989.
[31] J. Yuill, M. Zappe, D. Denning, and F. Feer, "Honeyfiles: Deceptive files for intrusion detection," in *Proc. IEEE Workshop Inf. Assurance*, Jun. 2004, pp. 116–122.
[32] B. Whitham, "Canary files: Generating fake files to detect critical data loss from complex computer networks," in *Proc. 2nd Int. Conf. Cyber Secur. Cyber Peacefare Digit. Forensic*, 2013, pp. 170–179.
[33] A. Cenys, D. Rainys, L. Radvilavius, and N. Gotanin, "Implementation of honeytoken module in DSMS Oracle 9iR2 enterprise edition for internal malicious activity detection," in *Proc. Conf. Detection Intrusions Malware Vulnerability Assessment*, Jul. 2005.
[34] S. J. Stolfo, "Software decoys for insider threat," in *Proc. 7th ACM Symp. Inf. Comput. Commun. Secur.*, May 2012, pp. 93–94.
[35] B. Bowen, S. Hershkop, A. Keromytis, and S. Stolfo, "Baiting inside attackers using decoy documents," in *Proc. Int. Conf. Secur. Privacy Commun. Netw.*, 2009, pp. 51–70.
[36] J. Voris, J. Jermyn, A. D. Keromytis, S. J. Stolfo, "Bait and snitch: Defending computer systems with decoys," in *Proc. Cyber Infrastructure Protection Conf. Strategic Stud. Inst.*, Sep. 2013.
[37] J. Voris, N. Boggs, and S. J. Stolfo, "Lost in translation: Improving decoy documents via automated translation," in *Proc. IEEE Symp. Secur. Privacy Workshops*, May 2012, pp. 129–133.
[38] N. Provos and T. Holz, *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*, 1st ed. Reading, MA, USA: Addison-Wesley, Jul. 2007.
[39] Y. Adachi and Y. Oyama, "Malware analysis system using process-level virtualization," in *Proc. IEEE Symp. Comput. Commun.*, Jul. 2009, pp. 550–556.
[40] I. Alberdi, E. Philippe, O. Vincent, and N. Kaaniche, "Shark: Spy honeypot with advanced redirection kit," in *Proc. IEEE Workshop Monitoring Attack Detection Mitigation*, Nov. 2007, pp. 47–52.
[41] The Honeynet Project, "Know your enemy: Sebek." [Online]. Available: http://www.honeynet.org
[42] L. Spitzner, "Honeypots: Catching the insider threat," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2003, pp. 170–179.
[43] Y. M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King, "Automated web patrol with strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities," presented at the *Proc. Symp. Netw. Distrib. Syst. Secur.*, San Diego, CA, USA, Feb. 2006.
[44] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna, "Escape from Monkey Island: Evading high-interaction honeyclients," in *Proc. 8th Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, Jul. 2011, pp. 124–143.
[45] B. Dolan-Gavitt and Y. Nadji, "See no evil: Evasions in honeymonkey systems," Tech. Rep. New York University, May 2010.
[46] K. Wilhoit, "Havex, It's down with OPC." [Online]. Available: https://www.fireeye.com/blog/threat-research/2014/07/havex-its-down-with-opc.html
[47] D. Hentunen and A. Tikkanen, "Havex hunts for ICS/SCADA systems." [Online]. Available: https://www.f-secure.com/weblog/archives/00002718.html

Julian L. Rrushi received the PhD degree in computer science from the University of Milan, in 2009, and the postdoctorate degree in computer science from the University of New Brunswick, in 2011. He is a tenure-track assistant professor of computer science and engineering with Oakland University, Michigan. He works at the intersection of defensive cyber deception and industrial control systems security. He worked for several years in industry as a vulnerability researcher. He has ethically discovered and analyzed numerous 0-day vulnerabilities in industrial control systems, which have been reported to the affected vendors. He has ethically broken and mitigated proprietary cryptographic algorithms that resided in the original design of new industrial devices destined for the market. He is a member of the IEEE and ACM.