A Product Machine Model for Anomaly Detection of Interposition Attacks on Cyber-Physical Systems

Carlo Bellettini and Julian L. Rrushi

Abstract In this paper we propose an anomaly intrusion detection model based on shuffle operation and product machines targeting persistent interposition attacks on control systems. These attacks actually are undetectable by the most advanced system call monitors as they issue no system calls and are stealthy enough to transfer control to hijacked library functions without letting their saved instruction pointers get stored on stack. We exploit the fact that implementations of control protocols running in control systems, which in turn are attached to physical systems such as power plants and electrical substations, exhibit strong regularities in terms of sequences of function calls and system calls issued during protocol transactions. The main idea behind the proposed approach is to introduce NULL function calls within a Modbus binary and to apply the shuffle operation between them and existing function calls. We then devise and implement a product machine capable of recognizing the shuffle representation of function call and system call regularities. A sensor uses a unidirectional interprocess communication channel based on shared memory to receive profile data from a Modbus process, and subsequently submits them to the product machine. We describe an experimental evaluation of our model on an ARM-based Modbus device and demonstrate that the proposed model overcomes the limitations of state of the art approaches with regard to detection of persistent interposition attacks on control systems.

1 Introduction

With the advent of low cost computing the control systems industry is replacing its proprietary legacy hardware with state of the art devices. The interconnectivity of control systems has transitioned drastically from minimal communications over

Carlo Bellettini and Julian L. Rrushi
Università degli Studi di Milano, Dipartimento di Informatica e Comunicazione, Via Comelico 39/41, 20135 Milano, Italy, e-mail: {carlo.bellettini, julian.rrushi1}@unimi.it

dedicated serial-line channels to Ethernet TCP/IP networks connecting control systems to each-other and often to the enterprise network and/or Internet. Proprietary communication protocols and operating systems have been in part replaced with open standards such as IEC 61850, DNP3, Modbus, IEC 60870-5, etc., and modern operating systems such as Windows CE or real-time variants of Linux, respectively. The actually high connectivity of control systems along with their use of standard technology expose control networks to a variety of network attack vectors. In fact several studies have shown that control systems are subject to various kinds of vulnerability relying in their data, security administration, architecture, networks, and platforms[18]. In particular, low-level coding vulnerabilities exploitable by memory corruption attacks have been found to be widespread in control system code. Unclassified case studies include a heap overflow in Inter Control Center Protocol (ICCP)[19], a heap overflow in LiveData Protocol Server [8], faulty mappings between protocol elements, i.e. handles and protocol data unit addresses, and main memory addresses in OLE for Process Control (OPC)[13][20], and faulty mappings between data items in a protocol data unit (PDU) as addressed by Modbus and the memory locations where those data items are stored[3].
In this paper we provide an anomaly intrusion detection technique based on concepts which we borrowed and adapted from automata theoretic models of parallel computation, namely shuffle operations and product machines[5, 9]. The proposed technique has been devised to detect persistent interposition attacks[15] and is carried out through sensor agents placed in control systems. These sensor agents gather execution data from a process to be protected and apply an automata–based recognition algorithm for the purpose of determining whether an intrusion is taking place. In our opinion the very first line of defense from network attacks on control systems should be some host-based intrusion prevention approach such as the one provided in [4]. Nevertheless, counting for the highly sensitive role played by control systems in monitoring and controlling critical infrastructures, additional security levels are necessary for countering network attacks. Moving along this line the proposed intrusion detection technique is intended as an additional layer of defense.
The remaining of this paper is organized as follows. Section 2 provides the motivations behind our contribution and describes representative related work. Section 3 provides the main ideas behind the proposed intrusion detection model. Section 4 describes an experimental evaluation of this model on a Modbus device[12] running an embedded Linux operation system. Section 5 summarizes our contribution and concludes the paper.

2 Motivations and Related Work

Anomaly intrusion detection systems (IDS) based on system call information rely on monitoring interactions between a process in user land and an underlying operating system kernel. A process to be protected is sent regular data in input several times and in isolation, i.e. off line. All system calls which are issued to the kernel

by this process are analyzed for the purpose of building a normal execution profile. In literature such an operation is often referred to as the learning phase. In monitoring mode an anomaly IDS starts monitoring system calls issued by a process to be protected and compares. Such an IDS compares the information extracted from monitored system calls with the information extracted from system calls issued during the learning phase. If deviations are identified, then the IDS concludes that an intrusion has taken place, hence the detection type – anomaly detection.

Thus, a process–to–kernel interaction through system calls is used as a mechanism for profiling a process to be protected. Along their way system call based IDS have had various limitations in extracting information from system calls in a proper and thorough way. Nevertheless, modern system call based IDS are quite powerful in exploiting system call information. If we assume an ideal system call based IDS, i.e. an IDS which has the capabilities of capturing the whole context of process–to–kernel interactions through system calls, could we state that we have in hand an IDS which is capable of detecting all known attacks on a protected process ? The answer to our question comes directly from the research work discussed in [15] that describes an attack technique which has been demonstrated to be capable of evading powerful system call based IDS. Authors refer to such a technique as persistent interposition attack, and we verified its offensive capabilities on control systems.

A persistent interposition attack relies upon an initial shellcode injection attack. After gaining execution flow control, injected shellcode corrupts pointer tables such as the global offset table (GOT) in ELF, virtual table pointers in C++ code, or any specific support for plug-ins and modules. The aforementioned corruption is carried out in such a way that attack code interposes itself between a target process and *write()* & *read()* functions of a C library. A persistent interposition attack intercepts in a man in the middle (MITM) style and subsequently modifies either all or selected parts of data read and/or written by a target process. Taking into account that a persistent interposition attack only modifies the I/O data stream of a target process, and does so by limiting itself not to issue any system calls or corrupt any data stored on stack, in traditional computer systems it may not be sufficient for attackers to achieve their objectives. That said, from our evaluation of persistent interposition attacks as applied on control systems results that such attacks are extremely powerful and fully sufficient to achieve attacker goals. In fact relevant objectives of attacks on control systems center around gaining control over control protocol frames holding commands to underlying critical infrastructure utilities or status information to be processed by a master station.

Obviously all those IDS which rely only on sequences of system calls issued by a given process have no means of detecting a persistent interposition attack since the latter absolutely does not cause any changes to sequences of system calls. Thus, a persistent interposition attack issues no system calls on a target platform. The VtPath model[6] goes beyond sequences of system calls and points towards call stack information in conjunction with system call information. As a system call is issued by a given process, VtPath extracts the system call name and the value of instruction pointer (IP) register. Further, VtPath extracts from the stack all routine return addresses preliminarily saved on stack and puts them into what authors refer to as

a virtual stack list. The value of IP register is then added to the end of the virtual stack list. For the purpose of characterizing a transition from a system call A to another system call B VtPath employs virtual stack lists. It uses them to build a logical virtual path from A to B that abstracts execution from the moment the process issued system call A to the moment the process issued system call B. If during the monitoring phase VtPath cannot construct the virtual stack list, then it assumes that a successful attack has corrupted return addresses that were stored on it. This fact is referred to as a stack anomaly. If an address is missing in the virtual stack list, then we have a return address anomaly. If the extracted value of IP register does not correspond to the system call name, then we have a system call anomaly. If there are any deviations in the virtual paths between two system calls, then we have a virtual path anomaly.

The work in [7] also combines system call information with call stack information defining an observation as a vector of a system call number along with the return addresses present on the stack in the moment this system call is issued. Executions then are thought as arbitrary-length sequences of observations and are used to create the so-called execution graph characterizing the behavior of a process to be protected.

In both the VtPath model and the execution graph model we can observe that the factors which create some kind of virtual fence to prevent injected shellcode from achieving attack goals without being detected from the IDS, i.e. call stack configuration and systems calls, are situated in the offensive space of injected shellcode itself. In fact nothing prevents injected shellcode from writing to a corrupted stack in order to restore return addresses before a system call is issued, hence evade detection. Further, although injected shellcode cannot issue a system call itself since that way it would cause what VtPath refers to as a system call anomaly, the work in [10] has demonstrated that the injected shellcode could jump to existing legitimate code in order to have that code execute the system call for it, i.e. for injected shellcode. Memory locations and registers are corrupted in such a way that the execution control is returned to injected shellcode after the issuance of a system call.

We deem approaches such as VtPath model and execution graph model to be still capable of detecting system call issuing attacks on control systems if an additional observable factor is employed, namely transaction response time. We exploiting the fact that industrial control communications are supposed to be real-time. In a typical Modbus transaction, for instance, a master station sends a Modbus request protocol data unit (PDU) to a slave device. The slave device examines the request and is supposed to reply with either a Modbus Response PDU or Modbus Exception Response PDU within a time limit which in most cases is in the range of just a few milliseconds. The response time of a slave device in normal transactions may slide between upper and lower time boundaries within the allowable response time. Let's take as an example a scenario where we have a sensor running on each slave device on a field. Let's assume that attackers acquire access to a process control network through a wireless node and start sending unauthenticated Modbus request PDU's to a target slave device on the field in order to exploit a memory corruption vulnerability in, say, a cryptographic routine.

If injected shellcode will try to reconstruct a corrupted stack each time it has existing

legitimate code issue a system call for it, then the sensor on the compromised field device will notice a considerable variation in the response time of the compromised device. This is due to the fact that reconstructing a call stack several times and regaining execution control from existing code requires a considerable amount of time. Powerful IDS models such VtPath and execution graph though don't have the instruments necessary for detecting persistent interposition attacks. As stated above persistent interposition attacks issue no system call. Further, after having intercepted and possibly modified function call arguments stored on stack, a persistent interposition attack uses a jump instruction to transfer execution control to the real *read()* or *write()* functions of the C library. In a system running on an ARM microprocessor[2] we had injected shellcode which transferred execution control to the real *read()* or *write()* functions by writing directly to R15 register or executing an unconditional branch instruction. This way the return address of the injected shellcode will not be saved on stack and no virtual path anomaly will take place. For more information on persistent interposition attacks please refer to [15].

3 A Product Machine Model for Anomaly Detection

The information extracted from user land by powerful system call monitors such as VtPath model and execution graph model for the purpose of creating a process profile consists of the value of program counter (PC) and return addresses saved on stack. We deem PC is quite a suitable mechanism for forcing attack code not to issue any system calls, at least not by itself. With regard to return addresses saved on stack we see several weaknesses which could allow an attacker to evade intrusion detection, namely:

- Information used for detection is stored on buffers which attack code can easily corrupt. In fact nothing prevents attack code from writing to selected locations on stack before a system call is issued and intrusion verifications are made.
- If attackers perform a deep analysis on target code and are patient enough to go through it, detection information itself, i.e. return addresses, may be calculated by attack code and used to restore a corrupted stack in order to cover attack traces in the moment of intrusion verification.

In fact a good part of implementations of control protocols and/or related libraries are provided by third party software companies and are accessible to everybody with enough financial support. A dedicated attacker could get the code of a target control protocol and analyze how stack is laid out immediately after gaining the execution control of a target process by transferring it to injected shellcode. Function call paths could also be analyzed as they directly influence stack layout. Further, none of the information employed by other models for intrusion detection is actually usable in detecting persistent interposition attacks which limit themselves to intercepting and modifying the I/O data stream of a target process without issuing any system calls. The proposed detection model aims at avoiding these weaknesses.

On one hand the approach obfuscates legitimate function call paths, thus invalidates attacker's knowledge required for evasion. On the other hand it prevents attack code from corrupting profile data generated by a monitored process. We first explain the concepts of our anomaly detection model. Then in the other section we describe the application of the proposed model to ARM-based devices with the premise that it is straightforward to do the same on control devices equipped with other embedded CPU architectures.

In the proposed model the information used as a basic building block in the activity of creating the profile of a process to be protected consists of the memory address of an instruction which issues a call to a defined function and the memory address where execution control is transferred shortly thereafter. Thus, when execution flow moves from one function A to another function or code block B, we're concerned with extracting the address of the instruction in A which issued the function call and the address in B where execution control is transferred. As in different executions a parent process along with any child processes it forks could be loaded at different virtual memory locations, we use PC-relative addresses for identifying the memory locations in functions A and B where a function call is issued and where subsequently execution control is transferred, respectively. CPU architectures have a predefined register which acts as a PC. PC-relative addresses do not change in different runs, therefore they can be reliably used as profile data. In the proposed anomaly detection model we consider the PC-relative address of a memory location where execution flow is being transferred as a result of a function call.

That relative address in reality is an offset from PC in the moment a function call is issued. If we are working on a l-bit architecture and for the sake of simplicity assume that respective memory addresses will consist of l bits, by using PC-relative addresses we shrink the $2l$ bits needed as profile data, i.e. l bits of the address where a function call issuing instruction is stored + l bits of the address where execution flow is transferred, into only l bits of a PC-relative address. Thus, a PC-relative address serves as some sort of logical binding between the memory address of an instruction which calls a defined function and the memory address of an instruction to which execution control is transferred. If in formal language notation we define a letter as a PC-relative address, then the alphabet containing all letters which are of our interest, i.e. PC-relative addresses, may be defined as follows:

$$\Sigma = \{x \,/\, alph(x) = \{0,1\}, |x| = l\}$$

where $alph(x)$ denotes the symbols which appear in a given letter, and $|x|$ denotes a letter length in terms of number of symbols which appear in it along with their frequency.

If we start monitoring a process, extract a PC-relative address, where execution control is transferred, from each function call issuing instruction actually executed, and concatenate those PC-relative addresses, i.e. letters in formal language notation, in their order of appearance, then we get a word which characterizes important aspects of a process execution, namely what we refer to as inter-function transfer paths. At this point we could extend the concept of profile data to include the mem-

ory address of an instruction which issues a system call along with the system call number of a service requested to an operating system. The memory address of an instruction which issues a system call could still be relative to the value which PC register had at some predefined entry point in the beginning of process execution. Further, the system call number could be padded to l bits in order to give it an appearance which complies with how we define a letter. Thus, a given inter-function transfer path is an ordered sequence of l-bit values taken as a function call is issued or as an operating system service is requested through a system call.

Let Σ denote an alphabet, i.e. a finite set of symbols or letters. If we don't consider predefined address space ranges where a process may be loaded, then the set of all possible arbitrary behaviors of a given process running on a l-bit architecture is characterized by the set of words, i.e. letter concatenations, as shown in the following definition:

$$\Delta = (\cup_{j=1}^{2^l} x_j)^*$$

where U denotes the union operator, x_j is an arbitrary letter, i.e. $x_j \varepsilon \Sigma$, and $*$ denotes the Kleene closure, i.e. Kleene star operation.

At this point we decide to introduce diversity into legitimate inter-function transfer paths of a process to be protected. We do so by inserting some NULL functions into the code of that process. NULL functions are inserted in such a way that they preserve the correct computability of a given program. For instance, if before inserting the NULL functions, say *w()* and *v()* into a program which has three functions such as *a()*, *b()* and *c()*, and where *a()* calls both *b()* and *c()* in that order, then an instance of a valid sequence of functions where execution flow passes through as a process proceeds with its execution would be:

$$\{w, v, a, v, a, b, a, c, v\}$$

As a result of a correct insertion of NULL functions into a program to be protected, a new inter-function transfer path will be created upon the previous one. The insertion of NULL functions into a program generally may cause deviations from the logical bindings via PC-relative addresses which were already present in the program before it got instrumented. Let's refer to these deviations as jitter. The new inter-function transfer path then will be composed of those original logical bindings via PC-relative addresses as changed by jitter, interleaved with new logical bindings via PC-relative addresses. These new logical bindings are due to new function call issuing instructions inserted as a result of program instrumentation. NULL functions may be defined as having an arbitrary number of arguments and local variables with arbitrary lengths. Allocation of memory for these variables would contribute to further obfuscate the stack layout. Considering the motivation behind introducing function arguments and local variables in NULL functions, their values could be arbitrary as long as they respect the type of the variables to which they are assigned. In addition, it is also useful to insert NULL system calls into a program to be protected as well. Going back to our formal language parallelism, we may notice that the word which represents a given inter-function transfer path of an instrumented

program is the result of the shuffle operation between the word which represents the inter-function transfer path of the same program before getting instrumented as subjected to jitter, and the word created by concatenating the logical bindings of NULL functions inserted into the instrumented program.
At this point we've built the basis for discussing how to proceed with an application to anomaly detection of the concepts described above. According to our model we run an instrumented program on a control system to be protected by sending it regular PDUs, namely those frames which should be expected to be received by this control system when it will be operational in a PCN. While doing so we make sure that the control system in question is not under attack, such as for instance by testing it in isolated laboratory settings. As a process under observation replies to various normal PDUs, we extract information about the function calls along with system calls it issues. In order to facilitate construction of a product machine to be used to recognize legitimate process behaviors, we need to differentiate between PC-relative addresses in an inter-function transfer path which represent original function or system calls, and PC-relative addresses in that inter-function transfer path which represent NULL function or NULL system calls.
For such a purpose we first record only those PC-relative addresses which represent calls to functions found in the original version of a program to be protected. Thus, the inter-function transfer paths we obtain in this step are those inter-function transfer paths which could have been observed while learning the normal profile of an original program, but which have been altered by jitter. We then repeat the profile learning procedure, but this time we record all PC-relative addresses in each function or system call actually issued. As a result of such a learning phase we have in hand a set of inter-function transfer paths which characterize legitimate behaviors of a process to be protected. Furthermore, we know which PC-relative addresses in an inter-function transfer path are due to original function or system call issues, and which of them are due to NULL function or system call issues.

The language defined as:

$$\Gamma = \{y \; \varepsilon \; \Delta \; / \; y \; \textit{has been observed during learning phase}\}$$

defines all normal behaviors of an original program to be protected. Γ is a finite language since there is a defined finite number of different function call sequences which a process follows during different executions upon receipt of a finite set of input frames.

In general, if Θ is the language composed of words created by concatenating letters, i.e. PC-relative addresses, of NULL functions and NULL system calls, then the language Υ which defines all normal behaviors of a program instrumented in all possible ways is defined as:

$$\Upsilon = \Theta \; || \; \Gamma'$$

where Γ' is composed of words representing inter-function transfer paths in Γ as possibly altered by jitter.
Our anomaly detection model uses a product machine as a recognizer of normal behaviors, i.e. words representing inter-function transfer paths which have been ob-

served during the learning phases, of a process to be protected. While a shuffle automaton could instead have been used in our model since we use shuffle operation to affect the behavior of program, we deem a product machine to be more appropriate in our context since we do not use a shuffle closure operation. According to our model we build a first finite state machine for recognizing strings made of PC-relative addresses which represent original function or system calls as altered by jitter, and a second finite state machine for recognizing PC-relative addresses which represent NULL function calls or NULL system calls. The ultimate recognizer of legitimate behaviors of a protected process then is the product of these two finite state machines. During the monitoring phase complete words of PC-relative addresses are collected from a monitored process and fed to the product machine. If such a machine recognizes those words, then the monitored process is exhibiting normal behavior, otherwise our model deems that the execution control of a monitored process has been hijacked.

4 Experimental Evaluation and Technical Details

Fig. 1 depicts a cyber-physical system where an experimental evaluation of our anomaly detection model was carried out. Note that the physical system, i.e. a nuclear power plant in our case, is simulated. The experiments were carried out on a Modbus device based on a 32-bit ARM microprocessor and running an embedded Linux operation system. More precisely, we worked on FreeModbus[21], which is a free implementation of the popular Modbus protocol especially targeted for embedded systems. For the purpose of experimentation we integrated FreeModbus into uClinux[1], an embedded operating system most suited for use in microcontrollers. In ARM microprocessors it is the register R15 which acts as PC, consequently while applying our model to an ARM architecture we use R15 as a base register to allow relative addressing. In fact register R15 at any moment holds a value which is the sum of the memory address of an instruction currently executing and 8 (well, in most cases, depends on implementation), but this fact has no affect on our assumption of logical binding between the memory address of a function call issuing instruction and the memory address holding an instruction to which execution control is subsequently transferred as a result of that function call.

In an application of our model to ARM we extract from each actually executed *BL* (branch with link) instruction, i.e. an ARM function call issuing instruction, what in ARM is referred to as a target address. The target address is a R15-relative address where execution control is being transferred, and represents exactly what in our model we refer to as logical binding. Further, we extract the address of each actually executed *SWI* instruction, i.e. an ARM system call issuing instruction, along with the system call number of a service requested to an operating system. Each extracted *SWI* address will be relative to R15 in a predefined entry point. The Modbus processes we experimented with acted as slaves, in the sense that they received requests and responded with responses. In fact in general configurations control

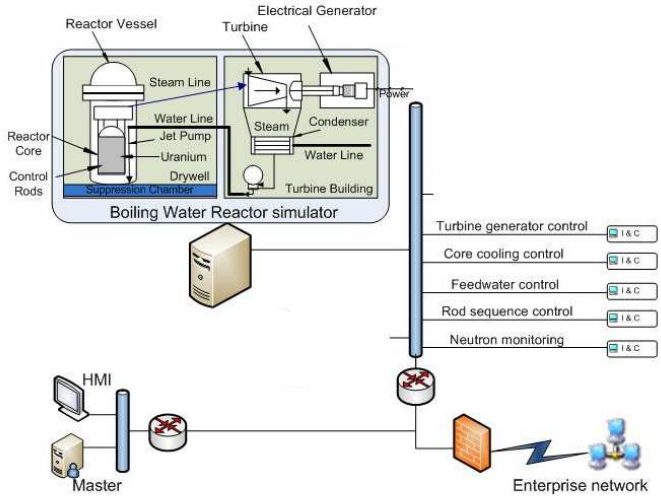

Fig. 1 An experimental testbed where the anomaly detection approach was practically evaluated

devices on a field act as slaves, except several cases in which they relay or even control protocol frames between other field devices. We coded a software agent which we used to extract data from a Modbus process as the latter proceeds with its execution. Further, we coded through the Concurrent Hierarchical State Machine language system[11] a small implementation of a product machine which we used as an intrusion detection mechanism following the proposed approach.

We started with a monitoring phase on FreeModbus tracing both its function calls and system calls in order to determine its actual inter-function transfer paths. We used testing utilities such as *modpoll* to send to the monitored process several Modbus request PDU's holding function codes of those which FreeModbus actually supports. We defined several NULL functions and a few NULL system calls which we inserted into the code of FreeModbus in such a way that the inter-function transfer paths newly formed were the result of a shuffle operation between R15-relative addresses of existing function calls and system calls as perturbed by jitter, and the inserted NULL function calls and NULL system calls. Rather than creating a profile of the whole FreeModbus code, we operated only on that part which actually executes during a Modbus transaction, which is from the moment a Modbus Request PDU is received from a socked till either a Modbus Response PDU or a Modbus Exception Response PDU is sent through a socket[12]. We used this defined interval as operational boundaries also during the monitoring phase.

Thus, the sensor agent starts to gather monitoring data from the Modbus process from the moment the latter receives a Modbus Request PDU. When a response is

generated by the Modbus process, the sensor agent feeds to the product machine implementation the string of monitoring data gathered that far. After doing so the sensor agent starts from the beginning. We set up an interprocess communication channel to allow the Modbus process send monitoring data to the sensor agent. The instrumentation of FreeModbus allowed us to also enable the Modbus process to send monitoring data to the sensor agent. We used shared memory as an interprocess communication mechanism. It is the sensor agent which creates a shared memory segment by issuing a *shmget()* system call. The shared memory permissions were such that the owner of the shared memory segment, i.e. the sensor agent, was allowed to attach to it in read mode, while others such as the Modbus process were allowed to attach to it in write mode. Thus, the Modbus process was supposed to write to the shared memory but not read from it, while the sensor agent was supposed to read from memory but not write to it. We used semaphores to synchronize the passage of monitoring data from the Modbus process to the sensor agent.
The intervention which enabled FreeModbus to send data to the sensor agent consisted in locating each *BL* instructions and subsequently inserting instrumentation instructions right before each one of them in order to extract their R15-relative *target_address* and write it to the shared memory so the sensor agent may grab it immediately. Further, we used the kernel to retrieve both the relative return address after a system call has been issued and the number identifying the service requested to the operating system. The defensive characteristics of our anomaly detection model which overcome the limitations of other powerful system call monitors described in previous sections are the following:

- As the sensor agent gathers monitoring data from the Modbus process throughout a transaction, an attacker cannot write valid data to the shared memory used for interprocess communication. This is due to the fact that an attacker does not know a valid inter-function transfer path as the original inter-function transfer paths have been shuffled with NULL function/system call R15-relative addresses.
- Attack code cannot overwrite monitoring data produced by legitimate FreeModbus instructions since due to process synchronization through semaphores those data are read by the sensor agent as soon as written on shared memory by FreeModbus.

Our anomaly detection model is devised to detect attacks which appear in an intra-process interposition form, therefore operating system security mechanisms should be properly employed to ensure that such attacks do not evolve into an inter-process interposition form. If a compromised Modbus process happens to run as a privileged account such as root in uClinux, then attack code could attach to a sensor process and take full control over its execution. In that case attackers could use DynInst API[14], which in earlier studies has turned out to be an easy to use and powerful attack tool[17]. An inter-process interposition form is also acquirable in the case both a Modbus process and a sensor process share user ID (uid) or group ID (gid). Therefore, a reasonable deployment of our anomaly detection model would consist in a Modbus process running as an unprivileged user, say *Modbus*, and in a sensor running as an unprivileged user, say *sensor*. Further, the uid and gid of user *sensor*

should be different than the uid and gid of user *Modbus*, respectively.
Let's see how our anomaly detection model behaves in front of a persistent interposition attack and an attack such as the one described in [10]. During the *initial exploit phase* phase of a persistent interposition attack a memory corruption vulnerability is used to transfer execution control to initial attack code. Such code is responsible for possibly downloading and storing bootstrap code, interposing that bootstrap code, and cleaning up any damage caused by the execution control hijacking. If the hijacking of execution control to the initial attack code is done through corruption of a return address or frame pointer stored on stack, our model is not likely to detect it since such a hijacking won't involve any of the *BL* instructions we keep under monitoring. If corruption of any function pointers within the address space of a target process is used to hijack execution control to the initial attack code, as it may be the case of a heap overflow or format string attack corrupting an entry in a function pointer table such as GOT, then our model will detect it. Considering that GOT in addition to ELF is also part of BFLT format[16], which in turn is used for formatting uClinux executables, let's take an example in which an attacker corrupts a GOT pointer to a library function *f()*. Let an original inter-function transfer path produced by a given application of a shuffle operation be $\{h, s, g, l, t, w, y, m, n\}$, where *w* is the address of function *f()* relative to R15, *m* is the address of a system call issuing *SWI* instruction relative to R15 in a predefined entry point, and *n* is a system call number padded to 32 bits.
Since the 32-bit data value in GOT which pointed to *f()* has been corrupted with a value which points to injected code, when function *f()* is called by the Modbus process execution control is transferred to injected code. During such a hijacking the *BL* instruction which was supposed to transfer execution control to the *w* R15-relative address, i.e. call function *f()*, in fact transfers execution control to, say, *v* R15-relative address, i.e. calls injected code. The inter-function transfer path extracted during the monitoring phase from such a hijacked process would be $\{h, s, g, l, t, v, y, m, n\}$. When this inter-function transfer path is fed by a sensor to the respective product machine, that product machine will not recognize such an observed inter-function transfer path, causing the IDS system to visualize intrusion alarms on HMI.
Regardless of detection of any hijackings of execution control to initial attack code, our model results to be capable of detecting a persistent interposition attack during what authors in [15] refer to as *bootstrapping phase*. In fact during the *interposing bootstrap code step* of *initial exploit phase* initial attack code modifies one or more function pointers in order to interpose bootstrap code. The effects of corruption of function pointers are observed in *bootstrapping phase* when bootstrapping code is invoked during all *read* and *write* operations. As each of these invocations takes place, a sensor registers from the *BL* instruction issuing the call the R15-relative address of the instruction of bootstrapping code where execution control is transferred. Such an address will cause a deviation in the legitimate inter-function transfer path making it unrecognizable by the product machine. The same consideration holds for *operational phase* during which execution control is continuously hijacked, consequently causing deviations in observed inter-function transfer paths which in turn

will not be recognized by the product machine.

Attack code itself cannot execute system calls without being detected. In fact, as a Modbus process requests an operating system service the kernel registers the return address and system call number. Such information is received by the sensor which incorporates it into the inter-function transfer path observed during the monitoring phase throughout a given Modbus transaction. The presence of NULL system calls in a shuffled behavior of a Modbus process would require attack code to scan most of existing instructions one by one in order to identify the system call numbers used in them. Walking through the executable segment though requires considerable time and processing logic. Further, if attack code issues a system call by itself, then its return address will be stored on stack and will lead to an easy calculation of the address of *SWI* instruction which issued the system call. The offset of the address of *SWI* instruction which issued the system call with respect to the value of R15 in a predefined entry point would cause a deviation in the observed inter-function transfer path.

Going back to our inter-function transfer path example and assuming a best case scenario for attackers, i.e. they make it to properly replay the NULL system call numbers, if such an offset is different than m, say q, then the following unrecognizable bahaviour representation will be observed on a compromised process: $\{h, s, g, l, t, w, y, q, n\}$. Thus, attackers would still need to issue system calls through *SWI* instructions in existing code and apply the IDS evasion techniques provided in [10] in order to regain execution control. Nevertheless, while reaching a returning point existing executing code may issue further function calls or even system calls, consequently additional R15-relative addresses or padded system call numbers may be inserted into the observed inter-function transfer path rendering it unrecognizable by the product machine applying the proposed intrusion detection model. The main techniques for regaining execution control according to [10] are modifying function pointers and return addresses stored on stack.

Modifying function pointers would immediately cause variations in inter-function transfer paths, thus their detection is straightforward. On the other hand, walking through the stack and searching for suitable return address values to corrupt requires time. Further, the knowledge required for locating a stack frame suitable for regaining control is not available since the sequence of original function calls has been shuffled with NULL function calls. Thus, an attacker does not know the stack layout and has to learn it. All these steps necessary for evasion cause a delay in both the total response time of a field device under attack and the time which passes between reception of consecutive R15-relative addresses in an observed inter-function transfer path. Our anomaly detection model takes into account these two different but inter-related delays for realizing that an evasion has potentially happened at a given field device.

The performance overhead induced by our anomaly detection model on a control system is dependent on the number of NULL function calls and NULL system calls inserted into a binary corresponding to an implementation of a control protocol. Inserting into FreeModbus from 2 to 4 NULL system calls and a number of NULL function calls which is a quarter of the overall number of function calls in the origi-

nal FreeModbus code induces a performance penalty of 6% over the total response time of a Modbus process during a typical transaction. Such a performance cost is due to gathering inter-function transfer paths and processing them through a product machine.

5 Conclusion

In this paper we provide an anomaly detection model based on shuffle operations and product machines. The main idea behind the proposed model consists of inserting into a process what we refer to as NULL functions and NULL system calls. In this model we represent each issuance of a function call or a system call as an R15-relative address. We then apply a shuffle operation between R15-relative addresses of existing function calls and system calls, and NULL function calls and NULL system calls. With regard to the decision mechanism we employ a product machine which recognizes the result of the shuffle operation described above. If such an automaton does not recognize any single string of data gathered from a process running in a control system, then the proposed model deems that an attack is taking place in the control system in question. As a conclusion, we demonstrate the efficiency and feasibility of the proposed anomaly intrusion detection model. We show that it overcomes the limitations of state of the art system call monitors by providing an experimental evaluation on a Modbus ARM-based device running an embedded version of the Linux operating system.

Acknowledgment

We would like to thank Prof. Stefano Crespi Reghizzi of the Politecnico di Milano for useful discussions and suggestions that he provided throughout the implementation of this work. Julian Rrushi was supported in part on a doctoral scholarship from the Università degli Studi di Milano, and in part on a research scholarship from $(ISC)^2$. Opinions, findings, and conclusions expressed in this paper are those of the authors only.

References

1. Albanowski K, Dionne DJ Embedded Linux/Microcontroller Project. http://www.uclinux.org/pub/uClinux/ports/arm7tdmi/ Cited 14 Jan 2008
2. ARM Limited, ARM Technical Reference Manuals. http://www.arm.com/documentation/ARMProcessor_Cores/ Cited 14 Jan 2008
3. Bellettini C, Rrushi JL (2007) Vulnerability Analysis of SCADA Protocol Binaries through Detection of Memory Access Taintedness. In: Proceedings of the 8th IEEE SMC Information

Assurance Workshop, United States Military Academy, West Point, New York, USA, pp. 341–348
4. Chen Sh, Xu J, Nakka N, Kalbarczyk Z, Iyer RK (2005) Defeating Memory Corruption Attacks via Pointer Taintedness Detection. In: Proceedings of IEEE International Conference on Dependable Systems and Networks, Japan
5. Diekert V, Mètivier Y (1997) Partial commutation and traces, Handbook on Formal Languages, 3rd volume, Springer: 457–533
6. Feng H, Kolesnikov O.M., Fogla P., Lee W, Gong W (2003) Anomaly Detection Using Call Stack Information. In: Proceedings of IEEE Symposium on Security and Privacy, Oakland, California, USA
7. Gao D, Reiter MK, Song D (2004) Gray-box extraction of execution graphs for anomaly detection. In: ACM CCS
8. iDefense (2007) Livedata protocol server heap overflow vulnerability. http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=523 Cited 14 Jan 2008
9. Jedrzejowicz J (1999) Structural properties of shuffle automata. In: Grammars 2, number 1, 1999.
10. Kruegel C, Kirda E, Mutz D, Robertson W, Vigna G (2005) Automating mimicry attacks using static binary analysis. In: Proceedings of 14th Usenix Security Symposium, Baltimore, USA
11. Lucas PJ, Riccardi F Concurrent Hierarchical State Machine. http://chsm.sourceforge.org Cited 14 Jan 2008
12. Modbus Organization (2004) MODBUS Application Protocol Specification V 1.1a. http://www.modbus.org/docs/Modbus_Application_Protocol_V1_1a.pdf Cited 14 Jan 2008
13. Mora L (2007) OPC server security considerations. In: Proceedings of SCADA Security Scientific Symposium, Miami, Florida, USA
14. Miller BP, Christodorescu M, Iverson R, Kosar T, Mirgordskii A, Popovici F (2001) Playing inside the black box: Using dynamic instrumentation to create security holes. In: Parallel Processing Letters
15. Parampalli C, Sekar R, Johnson R (2007) A Practical Mimicry Attack Against Powerful System-Call Monitors. Technical Report SECLAB07-01, Secure Systems Laboratory, Stony Brook University, USA
16. Peacock C uClinux - BFLT Binary Flat Format. http://www.beyondlogic.org/uClinux/bflt.htm Cited 14 Jan 2008
17. Rrushi JL, Rosti E (2005) Function Call Tracing Attacks to Kerberos 5. A presentation of Conference on Detection of Intrusions and Malware & Vulnerability Assessment, Vienna, Austria
18. Stamp J, Dillinger J, Young W, DePoy J (2003) Common Vulnerabilities in Critical Infrastructure Control Systems. A Technical Report of Sandia National Laboratories, Albuquerque, NM
19. US–CERT (2006) LiveData ICCP Server heap buffer overflow vulnerability. Vulnerability note VU#190617
20. US–CERT (2007) Takebishi Electric DeviceXPlorer OPC Server fails to properly validate OPC server handles. Vulnerability note VU#926551
21. Walter C FreeMODBUS library". http://www.freemodbus.org/ Cited 14 Jan 2008