Improving Resilience by Deploying Permuted Code onto Physically Unclonable Unique Processors

Andreas Aßmuth[†], Paul Cockshott[*], Jana Kipke[†], Karen Renaud[*], Lewis Mackenzie[*], Wim Vanderbauwhede[*]
Matthias Söllner[†], Tilo Fischer[†], George Weir[‡]
[†]University of Applied Sciences OTH Amberg-Weiden. Corresponding Author: a.assmuth@oth-aw.de
[*]University of Glasgow
[‡]University of Strathclyde

***Abstract*—Industrial control systems (ICSs) are, at present, extremely vulnerable to cyber attack because they are homogenous and interconnected. Mitigating solutions are urgently required because systems breaches can feasibly lead to fatalities. In this paper we propose the deployment of permuted code onto Physically Unclonable Unique Processors in order to resist common cyber attacks. We present our proposal and explain how it would resist attacks from hostile agents.**

I. Introduction

Cyber attacks on traditional information technology (IT) systems steal or destroy information, which is damaging enough, and has financial implications. Nowadays, Industrial Control Systems (ICSs) are threatened by attacks similar to those against traditional IT used in companies to support office work. Attacks on ICSs can lead to tangible, physical side effects, which could potentially be cataclysmic.

The reason for this is that ICSs measure and control physical processes. Examples are the automation of processes in factories, the control of large systems in industrial production, or coordination of critical infrastructures. The use of Stuxnet to sabotage the Iranian Nuclear programme [1] revealed the latent vulnerabilities of ICSs. An unknown agent introduced malware into the system by plugging in a thumb drive which then installed the malware. It propagated itself across the plant and sabotaged the plant's functioning so that severe damage resulted.

Since ICSs run many kinds of critical systems it is vital to find a way to make these systems more resilient to cyber attack. The Stuxnet attack did not result in any fatalities, but one can easily imagine attacks on other systems leading to multiple fatalities were hostile agents to be able to take control over them.

In this paper we propose combining two techniques to address the threat: (1) the use of physically unencodable unique processors, and (2) permuted operator tables. The former ensures that software will not run on any but the specific processor it was produced for. The latter ensures that even if one particular system is compromised, it would not be possible for the code to propagate itself with ease, which has been the case for malware thus far.

II. Related Work

A. Attacks on Industrial Control Systems (ICSs)

Cyber attackers targeting ICSs normally face two different scenarios [2]:

(a) The ICS is *directly connected* to the Internet. In this case, the device is usually attacked directly. This means attackers try to exploit vulnerabilities and weaknesses in the *Operating System* (OS) or maybe the configuration interface which is accessible via a web server running on the device itself. Other threats are similar to those on any other device that has direct internet access, like, for example, denial of service attacks (DoS).

(b) If the ICS *cannot be reached over the Internet*, attackers try to infiltrate the traditional office IT of the target company in order to build sort of a beach head. This is achieved by common attack techniques, e.g. by spear-phishing or even social engineering. From here, the attacker gathers more information about the network and the attached devices, tries to find vulnerabilities and exploits these to attack ICSs. The already mentioned Stuxnet worm is a primary example of this scenario [3].

It is not trivial to resist attacks against ICSs since the techniques used to defend against traditional attacks on office IT systems are not fit for purpose due to the special requirements of ICSs. ICSs are generally time-critical and must meet hard or soft real-time requirements [4]. The outage of ICSs is unacceptable because this would halt production. Forced errors in an ICS can result in malfunctions of robots, conveyor belts etc. which could lead to severe injury or even the loss of life. Therefore, ICSs must have some kind of fault tolerance. ICSs can also be used in environments with restricted accessibility, e.g. *in vacuo* or when using toxic gases, and this complicates physical access to devices. ICS-based plants are usually designed to work for years without interruption, which means that the systems used to control the production process are expected to function properly for at least 15 to 20 years.

B. Physical Unclonable Functions

Software developed for a particular OS will run on any other system with that OS, regardless of the underlying hardware. Software can be updated by a manufacturer, and such updates propagated to all systems without individual installation. Attackers exploit this very characteristic to install their malware on entire plants. Any authentication of the entity installing updates is achieved by software, usually using passwords. Given the ease with which attackers gain access to passwords they do not constitute a significant barrier. Some systems make use of hardware authentication, but these are easily compromised if an attacker gains access to a device.

Having an individual inimitable feature to unmistakeably identify individual devices would address this vulnerability. Such devices would also be able to distinguish between rightful access by parties with a profound knowledge of these features and unauthorised connection attempts by hostile agents. One possibility is to equip devices with *physical unclonable functions* (PUFs). PUFs are hardware entities that use device-specific features (based on natural variations in the production process) to generate unpredictable device-specific responses to incoming challenges. These challenge-response pairs are unique for each PUF, since the production parameters are fragile and impractical to replicate. It is also possible to design PUFs in such a way that simulation of the behaviour of a given PUF on a general purpose computer is impossible.

Similar to biometrics, this inherent structure, as the foundation of any PUF-scheme, can act as a hardware fingerprint of single electronic devices or even distinct chips. PUFs are functions in that they accept inputs and use their unique fingerprints to derive unpredictable device-specific responses to incoming challenges. Since this response derivation should not be invertible. Pappu *et al.* [5] suggested the name *Physical One-Way Functions* (POWF) for these, and Gassend [6] called them *Physically Random Functions* (PRF) since it should be impossible to predict responses to given challenges without using the PUF itself.

The challenge-response behaviour of any PUF could be described by a function $F(C) = R$ that maps a suitable challenge C onto a unique and unpredictable response R. For security applications this function must be mathematically unclonable. This means no attacker should be able to derive a function F' that describes the mapping for the used set of challenge-response pairs (C, R). Depending on the scenario and the exact category of PUF used, an attacker is considered to be able to gain some knowledge about the PUF. For example, knowledge of the construction outlay or a limited subset of corresponding challenge-response pairs of this, or a similar PUF, without being able to derive an appropriate function reliably to map any other challenge-response pairs.

Furthermore, PUFs can be constructed in such a way that reverse engineering, or any attempt to circumvent the usual challenge-response procedure, will destroy the hardware fingerprint of the device. If an attacker gets hold of such a device their only way to approach the PUF would be to feed it challenges through the designated channels of the device.

C. Strong PUF, Physically Obfuscated Keys and Controlled PUFs

The number of challenges can vary from very large (so called "*strong PUF*") to rather small or even only one challenge-response pair ("POKs — *Physically Obfuscated Keys*"). Both categories share the concept of a secret derived from a physically uncloneable structure, however their possible usage and attack points differ drastically.

For strong PUFs, security emerges from a large set of challenge-response pairs, which cannot all be readout during a single access of limited duration nor be guessed at from any given subset. Using this feature, a database of challenge-response pairs to one PUF can, for example, authenticate a holder of this PUF-device as the only one who can successfully give the right response to a random challenge. One of the first secure authentication protocols based on this feature was given by optical PUFs [5]. Attacks on the PUF by creating lists of challenge-response pairs, however, become infeasible if the number of pairs becomes too high to create an exhaustive database in a realistic timespan. For use of strong PUFs in security applications it is especially necessary that these entities provide mappings that are not only physically but also mathematically unclonable for the used set of challenge-response pairs, since most successful attacks on strong PUFs are based on modelling attacks and more recently "*combine modelling attacks with extra information obtained from direct physical PUF measurements or from side channels*" [7].

A *Physically Obfuscated Key* [8], on the other hand, can substitute for a secret key stored in hardware. It is assumed that no attacker is able to circumvent the access protocol and directly read the response. If this can be granted, the secret key can be used as shared secret key to encrypt communication to another holder of this key, such as the manufacturer. In this scenario PUFs constitute a feasible alternative to ordinary secret keys stored on hardware, since the secret is not permanently present but is only generated on demand. Another advantage is the fact that many PUF concepts have a tamper-evident feature, causing the PUF to show or even be destroyed at any attempt to manipulate it or record responses circumventing the access protocol.

In combination with POKs, when the device-specific features are used to derive a secret key, so called *controlled PUFs* (CPUFs) become important, to prevent an attacker from directly reading out the POKs response. A CPUF, as introduced by Gassend *et al.* [6] and summarised by Maes *et al.* [8], is "*in fact a mode of operation for a PUF in combination with other (cryptographic) primitives*". The PUF is, preferably irremovably, connected to the primitive in a way that prohibits access to the PUF apart from well defined channels involving the cryptographic primitive. This can, for example, mean that all challenges are processed by a hash function prior to being fed into the PUF, thus preventing the attacker from exploiting readouts of specific challenges (the equivalent of chosen plaintext attacks). Another similar approach is to feed the response from the PUF, after error correction, into an cryptographic algorithm. Thereby the connection between challenge and response can be obfuscated and the potential of being unclonable both in hardware and in modelling can be enhanced by using CPUFs. A CPUF-scheme can be used as well to give "*multiple personalities*" (Maes *et al.* [8]) to a PUF by allowing the cryptographic algorithms to accept parameter as well. For example in combination with a state-of-the-art block cipher a PUF may process two inputs: an encryption key and a message. In this way a PUF may serve either as a storage that provides a unique and unpredictable device-specific secret key or as generator of messages, which are encoded using a different key. For this proposed application, POKs in a CPUF context provide a technique for embedding within a processor an oracle which manufactures unique cryptographic keys.

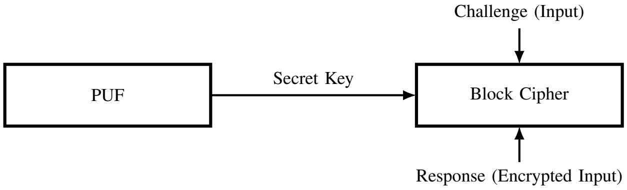

Fig. 1. Controlled POK

The construction of a CPUF from a POK, in combination with a block cipher, shows similarities with a strong PUF, in that the CPUF accepts inputs as challenges and gives responses, which can only be verified by a party in possession of the secret key. However, in comparison to a strong PUF, this construction solves the problems of storing a limited number of challenge-response pairs. Since only the secret key has to be stored securely a database of all such CPUF devices would need far less resources.

D. The problem of error correction

Because of the small variations in the structure, the stability of the responses, especially with respect to varying environmental factors during readout, has to be carefully taken into account. Maes *et al.* [8] and Katzenbeisser *et al.* [9] consider options and bounds for environmental factors in which variation between responses $R_1 \leftarrow F(C)$, $R_2 \leftarrow F(C)$ of a particular challenge on one single PUF ("*intra-distance*") are stable, while on the other hand the difference between responses $R_1 \leftarrow F_1(C)$, $R_2 \leftarrow F_2(C)$ to the same challenge on two different PUF instantiations ("*inter-distance*") becomes distinguishably large. In this context it is of major importance that efficient error correction is accounted for to compensate the intra-distance noise. Dodis *et al.* [10] describe error correction schemes especially for biometric input data. Their technique even allows error correction of a secret value w by a public information about w that, while not revealing w, still allows to restore w from an appropriately close value w'. By this, it is not necessary to store the data for error correction inside the PUF, which would oppose many advantages of the PUF concept. In addition, special protocols like the one described by Pappu *et al.* [5] are in themselves error tolerant. However, if such protocols can be used depends on the PUF application. Rührmair *et al.* [11] especially stressed, that perfect error correction in the secret key is needed in any application of POKs.

E. Permuted Code

Cyber attacks deploy attacks that are similar, in many ways, to attacks on human systems. Viruses and bacteria regularly seek to attack us, and our immune systems have been designed to resist such attacks. In nature, viruses tend to be species specific. A virus will infect cells by binding to particular proteins presented on the cell wall. This binding tends to be highly specific so that the binding area on the virus surface is only able to link to a specific sequence of amino acids. Because different host species present different amino acid sequences the ability of the virus to spread to new hosts is impaired.

In agriculture is it well known that cultivation of monocultures of single strains of a cultivar renders crops more vulnerable to attack by viruses and other pathogens. Similar phenomena occur in the computing world. The instruction set/operating system combinations that are most widely used are the most vulnerable to viruses.

Infection of a machine by a virus requires :

1) The writing of a tailored virus program exploiting a known vulnerability on a known category of physical or virtual machine.
2) Either the deliberate targeted introduction of the virus onto a specific machine or the spread of the virus through the general population of machines until it reaches the target machine.

At first virus spread was limited to machines like PCs with instruction sets/operating system combinations in widespread general use (e.g the Intel x86/MSDOS pairing). Critical infrastructure computing used a more diverse population of machines and OS types and was, in consequence, relatively immune to virus infection. This is changing with the greater use of standard *Instruction Set Processors* (ISPs) whether physical designs or virtual platforms.

The combination of a known OS and a small number of widely used ISP types means that safety- or mission-critical systems are much more vulnerable targets for infection. The infection can be deliberate and targeted as in the case of Stuxnet or the inadvertent result of the spread of viruses from the general computer population.

A computer virus is always a program written in some instruction set. As such, a virus that is targeted at Intel x86 ISP cannot infect machines based on ARM processors. If a sufficient number of different types of ISPs are in use, the spread of viruses is inhibited. But the design of a new ISP with a different instruction set is very costly so that there are relatively few in existence and each may have a large population of potentially infectable instantiations.

Our proposal is to introduce into the instruction cache fetch logic of an ISP, a permutation table. Suppose that the ISP uses byte codes. The normal cache load sequence goes something like:

```
fetch instruction at address X
SEQ
  check cache tags for X
  if X present return cache[tail(X)]
  else
  SEQ
    temp = mem[X]
    PAR
      cache[tail(x)] = temp
      return temp
```

We propose introducing an additional stage so that it becomes

```
fetch instruction at address X
SEQ
  check cache tags for X
  if X present return cache[tail(X)]
  else
  SEQ
    temp1 = mem[X]
    temp2= permtab[temp1]
    PAR
      cache[tail(x)] = temp2
      return temp2
```

The permutation phase performs a mapping from the instruction read in to the native code of the processor. The operation can be relatively easily carried by a 256-entry fast access table which maps the ordering of the source opcode set to the ordering of the native instruction set.

This mechanism allows a given hardware architecture to execute any one out of 256! different possible ISPs.

Machines would have unique instruction sets shared by no other system. This would effectively make them immune to viruses spreading in the general population. In principle, an opponent who knew the permutation table used in a particular instance could produce a highly targeted virus, but this would be prevented by the use of PUFs.

The processors in this family could be assembly language, but not binary language compatible with an existing family of machines. Provided a modified version of the assembler was available that took as an additional input an op-code permutation table, it would be possible to cross compile applications to the specialised machines a small extra cost. It would be important that the machines should not have a local compiler or interpreter that would enable them to generate code for their own permuted instruction set. This would prevent the spread of source code viruses.

The permutation mechanism would ensure that any attempts to use a buffer overflow attack would result in what was essentially a random sequence of numbers being interpreted as opcodes. The resulting code would either loop, or terminate on an illegal instruction almost immediately. The relative probability of these two occurences is obviously related to Chaitin's Ω constant [12], the probability that a random computer programme will halt. This is in principle uncomputable, but estimates for the leading digits of it as a binary fraction can be computed for specific machines. Calude *et al.* [13] report estimates for a simple register machine where they are able to prove that for a simple register machine of their specification the first 64 bits of Ω are

```
0000001000000100 0001100010000110
1000111111001011 1011101000010000
```

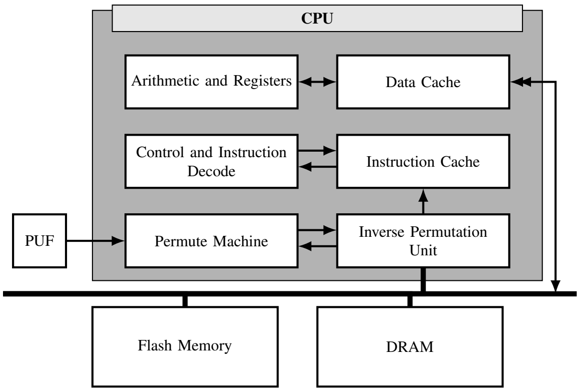

Fig. 2. Overal structure of the permuted code processor

This being for a machine with 10 possible opcodes, one of which is a halt instruction. This implies that the great majority of random programmes go into infinite loops. The halting probability will probably depend on the proportion of opcodes that are actually halt or illegal opcodes. It does not, from the standpoint of preventing deliberate malware or viruses, matter if the programme loops or crashes. But from the standpoint of reliability, this would still allow a buffer overflow vulnerability to cause a crash even though it would prevent an exploit. For industrial control purposes this would obviously protect against Stuxnet-type attacks; the attack is immediately visible, but it would not prevent attacks whose aim is to crash a machine.

Our proposed architecture is shown in Fig. 2. We assume that the data and code held in flash memory is in a strongly encrypted Linux filing system, encrypted using a key generated by the PUF on the chip. For environments where temperature made PUFs perhaps unreliable, this could be replaced by a key on an on-board ROM, but such a solution is second best for a number of reasons discussed earlier. The data held in the DRAM is unencrypted but the code is held in permuted form and as such would not be directly executable. The CPU is assumed to be a modified Harvard architecture at the low level with distinct data and instruction caches. The data cache has direct read/write access to the DRAM, but the instruction cache accesses the DRAM via an inverse permutation unit. Given a permuted opcode from the DRAM it translates it to a plain text opcode before passing it to the cache.

The actual inverse permutation is easily performed by a lookup table. Let us assume that the opcode field is b bits wide. A $2^b \times b$ bit RAM can perform this task. The contents of the inverse permutation table are initialised by a permutation machine fed by a pseudorandom bit stream from a PUF.

The precise algorithmic structure of the permutation machine is an implementation question, but it should follow a waterfall principle whereby if a single bit in the output of the PUF is on statistical grounds likely to change the majority of the permutation table positions. Here is an example of such an algorithm in C assuming $b = 8$.

```
char t1[256],t2[256];
pm(char *pufstream){
  register int i,j,k,t;
  for(i=0;i<256;i++) t1[i]=i;
  for(i=0,k=0;i<128;i++) {
    t=pufstream[k++];
    for(j=0;j<256;j++)t2[j^t]=t1[j];
    t=pufstream[k++];
    for(j=0;j<256;j++)t1[(j+t)%256]=t2[j];
  }
  for(j=0;j<256;j++)t2[t1[j]]=j;
}
```

The buffer `t1` is internal to the permutation machine the buffer `t2` is the inverse table in the permutation unit. A stream of 256 bytes is read from the PUF. At then end an inverse permutation table is in `t2`.

F. Related Permutation Work

Many researchers have studied the concept of *instruction-set randomization*. Here we consider practical work most similar to our proposal, and highlight how we intend to improve upon this existing work.

Barrantes *et al.* [14] describe a software-only technique for dynamic code randomization, to provide instruction set diversity in an attempt to thwart code injection attacks. They present a proof-of-concept implementation, called Randomized Instruction Set Emulation (RISE). This system is based on Valgrind, the Linux, open-source, dynamic binary translation framework. Executable code does not require recompilation or pre-execution modification. Instead RISE scrambles binary code as it is loaded by the OS, by XORing code bytes with a cached random number. The code is unscrambled on a per-instruction basis, during the instruction fetch into the Valgrind dynamic code cache. Then the unscrambled code may be executed directly by the underlying processor. RISE is demonstrated to work on a commodity x86 processor using the Linux OS. Note that the dynamic binary translation service provided by Valgrind (which implements the code randomization) is a layer on top of the OS, only for application code execution. The paper reports on tests using standard internet daemons (Apache web server, etc), and describes how RISE enables protection against buffer overflow attacks etc. This is a promising approach, but there are several disadvantages. The executed application code is slow because of the overhead of dynamic binary translation, which is implemented entirely in software. The OS and Valgrind DBT code run unscrambled on the stock hardware. Thus it might be possible to circumvent the scrambling process by injecting code into the DBT directly. Further, there are difficulties when applications store code in data sections (which are not scrambled). Explicit hard-coded workarounds are required to handle this problem.

Portokalidis *et al.* [15] use a similar approach to RISE, based on the Intel PIN dynamic binary rewriting tool. They use a database of keys, to randomize every loaded code image in a different way. Their work has lower runtime overhead than RISE, demonstrated by running the Apache web server and a database server.

Fechner *et al.* [16] report on preliminary work which is most related to our proposal. They sketch a technique to permute instruction opcodes at runtime, in the processor pipeline. Trusted code will be compiled by a customized toolchain which transforms the executable instructions according to the specified permutation. A virus which does not know the permutation therefore has instructions that are semantically nonsensical, and likely to cause a runtime failure such as a segmentation fault. The opcode permutation is implemented using a hardware lookup table. They give simple, preliminary results to explore how such a lookup table might be integrated into an instruction decode pipeline stage, using Field-Programmable Gate Array (FPGA) prototyping. To the best of our knowledge, they have not followed up on this initial work, to deploy the scheme in a realistic system.

Ichikawa *et al.* [17] describe an alternative instruction set randomization approach, specifically for embedded processors implemented in FPGA technology. They adopt a pragmatic approach, to minimize instruction decode complexity. They identify instructions that have the same bit-level format (e.g. `ADD r1,r2,r3` and `SUB r1,r2,r3`) and arrange a substitution cipher encoding between some subset of instructions sharing the same format. They evaluate this for several standard instruction sets, including MIPS and JVM bytecode. While this approach works well for RISC-like instruction sets with fixed length encodings and a few standard bit-level layouts, it would not be applicable to variable length x86 instructions.

Sun Microsystems patented a technique[18] which is similar in some respects. They propose a modifiable microcode dispatch table for a machine, such that a permutation of this table would result in a permuted semantics. The effect of executing the ADD instruction, could be for instance to dispatch the microcode of an XOR instruction. If the architecture already had a modifiable microcode table, then this would introduce no delay in the pipeline. If, however, the table was additional to the original design there would clearly be an extra pipeline stage.

We avoid this extra pipe-line stage, since most instruction fetches will go to the cache without creating a cache miss. Only on the relatively small percentage that result in a cache fail, will there be an extra clock cycle introduced into the cache load time.

III. Proposed Solution

In order to achieve the proposed properties, we present some ideas, how the goals could be reached. The proposals have to be further refined.

A. Manufacturing

The primary focus of this work is on processors for embedded systems, typically a System-on-Chip containing a CPU with additional custom hardware and off-chip RAM and NVM. The SoC can be implemented either as an ASIC (Application-specific integrated circuit) or as an FPGA. Typically, the embedded system will be created by a system integrator and the design of the SoC will be subcontracted to a design company. For the ASIC case, the design will typically be produced by a semiconductor manufacturing company; the FPGAs can simply be purchased from an FPGA manufacturer.

The system integrator builds and delivers the hardware system and equips it with an initial software version. Updates of the software will only be allowed by the manufacturer. The protocol of the software updating process will be explained later. The system is open to third party software, which has to be distributed by the manufacturer, in order to review it, check for and prevent against malicious code.

The firmware code has to be permuted at the manufacturer by using the secret key, generated by the PUF of the target device. This secret key has to be read out by the manufacturer in the manufacturing process before sealing the hardware and delivering it. This operation would typically be carried out as part of the automated testing stage.

B. Software Update Mechanism

The software update process has to be secured in several ways. Mutual authentication of the communication partners has to be realised. If authentication was successful the permuted software can be sent to the client, secured by a standardised secure transmission protocol such as TLS with preshared keys for example. The client stores the encrypted firmware in his flash memory. Before loading the firmware in the DRAM for execution decryption has to be done, in order to fill the DRAM with the permuted firmware code.

C. Command Interface Mechanism

We also propose a protocol for executing different commands on the target platform, for example health check, status retrieval or a target reboot. Similar to the software update process mutual authentication is necessary. Afterwards commands can be executed secured by a standard protocol like TLS. Figure 3 shows the idea.

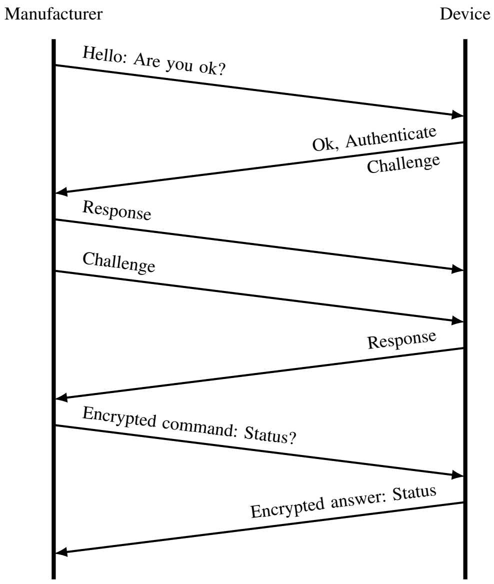

Fig. 3. Command Interface Protocol

IV. Attack Scenarios

Consider a group of ICS installations produced and managed by a single manufacturer who, by assumption, is using a hardware-software distribution mechanism modelled on the one described above. As in any such scenario, an intruder could attack the installations in many different ways and here we attempt to examine some of these without pretending to be exhaustive. We begin by observing that potential attacks can be usefully categorised in terms of their ambition. With this in mind, in order of decreasing seriousness, we will use the following terminology in the subsequent discussion. Each of these will be related to the applicable information security CIA principle.

A Category A attack is one which targets the entire group or a significant subgroup by attempting to infect many or all of the systems with the same malware. Often such an attack will employ a deferred payload, meaning that any damage intended is not inflicted immediately, thus giving the infection a chance to spread before it is detected. Viruses, worms and botnet type attacks all fall into this category. This is an attack on the *integrity* of the industrial control process.

A Category B attack is less ambitious and tries only to infect a single machine with malware. Such malware may be active or passive. Active malware aims to inflict damage but may also defer its payload to allow the attacker to choose the best moment to inflict maximum harm. Passive malware aims to provide the intruder with information or a control capability which may be used in the future. In all attacks in this category however, by assumption, there is no intention to spread the infection beyond the compromised device. This is an attack on the *integrity* of the industrial control process.

A Category C attack is less ambitious still and tries only opportunistically to crash a targeted device. Such an approach aims at corrupting existing code causing execution failure rather than attempting the more difficult task of introducing runnable malware. This is an attack on the *availability* of the industrial control process.

A Category D attack does not attempt to affect the operation of the devices at all, but only to eavesdrop on the system in such a way as to acquire information. This is an attack on the *confidentiality* of the industrial control process.

Attacks in any of these categories may try to exploit vulnerabilities using one of three broad mechanisms: by attacking the communications channel, the device site or the manufacturer site.

We note first that the permuted opcode system makes Category A attacks all but impossible because code can only run on a target ICS if its permutation is known and for this to be possible requires acquiring its secret key. Code for a given device will not run on any other and so it is not possible to engineer a virus or worm that can be spread between devices. There are two obvious ways an attacker might try to circumvent this inherent immunity. The first is to steal the manufacturer's secret key database and subsequently masquerade as the manufacturer so as to transmit individually permuted malware to all members of the group. This would also typically need the manufacturer's private authentication credentials as used to establish the secure link. The second is to inject malware into third party code before it is supplied to the manufacturer and therefore before it is permuted. If successful, such pre-infected code might be distributed to the whole group.

A Category B attack can also only succeed if a device's secret key is obtained, again along with the manufacturer's private security credentials. Since the key is never transmitted, a communications channel intrusion is not possible here. It is also not possible to extract the key from the ICS system itself without physically accessing and unsealing it: recall that the PUF mechanism presents the key externally only once, at manufacture time and is subsequently sealed. Thus a device site attack will not work either. The only feasible way to obtain the key required to attack an individual ICS is to gain access to the manufacturer's secret key database.

A Category C attack is easier for an attacker to carry out and can be conducted without gaining access to the secret key. All that is required is to corrupt code being distributed by the manufacturer to device sites, and to persuade the device to install this code, thereby putting it out of action. The only realistic approach is for the attacker to gain physical access to the manufacturer or device sites and then to damage code there or to deliberately corrupt third party code before it is sent to the manufacturer (although in the latter case this would presumably crash all devices to which the code was distributed).

Finally, a Category D attack requires only eavesdropping on the channel. Since all communications are encrypted, and both parties must mutually authenticate, we can assume that direct attacks that require decrypting traffic or masquerading as manufacturer or device will be infeasible unless an attack from one of the first two categories has already been successful. The challenge-response approach to authentication also protects against replay attacks and conventional encryption methods against message analysis of, for example, the transmitted code.

Summarising, it can be seen that the primary vulnerabilities are the manufacturer key database and the possibility of prior access to third-party code. Thus rigorous protection of the manufacturer database is crucial, along with the immediate post-production mechanism whereby the PUF-generated key is read out from a new device (prior to it being sealed) and entered into this database. Rigorous examination of third-party code *prior* to its distribution, will also obviously be essential possibly involving careful auditing and pre-testing on examplar systems at the manufacturer's site.

V. Conclusion

This paper proposed a hardware and software based mitigation mechanism for resisting malware. Our deployment platform of interest is ICSs that are particularly vulnerable to cyber attacks with maximum impact. We have not yet implemented this system, but propose it here in order to open a discourse on the feasibility of this solution as a mitigation strategy.

References

[1] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *Security & Privacy, IEEE*, vol. 9, no. 3, pp. 49–51, 2011.

[2] German Federal Office for Information Security. (2013, November) ICS Security Kompendium. [Online]. Available: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/ICS/ICS-Security_kompendium_pdf.html

[3] D. Kushner. (2013, February) The real story of stuxnet. [Online]. Available: http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet

[4] K. Stouffer, J. Falco, and K. Scarfone, "Guide to Industrial Control Systems (ICS) Security," National Institute of Standards and Technology (NIST), Tech. Rep. 800-82, June 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-82/SP800-82-final.pdf

[5] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions," *Science*, vol. 297, no. 5589, pp. 2026–2030, 2002.

[6] B. Gassend, D. Clarke, M. Van Dijk, and S. Devadas, "Controlled physical random functions," in *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*. IEEE, 2002, pp. 149–160.

[7] U. Rührmair and J. Sölter, "Puf modeling attacks: An introduction and overview," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[8] R. Maes and I. Verbauwhede, "Physically unclonable functions: A study on the state of the art and future research directions," in *Towards Hardware-Intrinsic Security*. Springer, 2010, pp. 3–37.

[9] S. Katzenbeisser, Ü. Kocabaş, V. Rožić, A.-R. Sadeghi, I. Verbauwhede, and C. Wachsmann, "Pufs: Myth, fact or busted? a security evaluation of physically unclonable functions (pufs) cast in silicon," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2012, pp. 283–301.

[10] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," *SIAM journal on computing*, vol. 38, no. 1, pp. 97–139, 2008.

[11] U. Rührmair and D. E. Holcomb, "Pufs at a glance," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[12] G. J. Chaitin, *Meta maths: The quest for Omega*. Atlantic London, 2006.

[13] C. S. Calude, M. J. Dinneen, C.-K. Shu *et al.*, "Computing a glimpse of randomness," *Experimental Mathematics*, vol. 11, no. 3, pp. 361–370, 2002.

[14] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized instruction set emulation to disrupt binary code injection attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*. ACM, 2003, pp. 281–289.

[15] G. Portokalidis and A. D. Keromytis, "Fast and practical instruction-set randomization for commodity systems," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 41–48.

[16] B. Fechner, J. Keller, and A. Wohlfeld, "Web server protection by customized instruction set encoding," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 5–pp.

[17] S. Ichikawa, T. Sawada, and H. Hisashi, "Diversification of processors based on redundancy in instruction set," *IEICE transactions on fundamentals of electronics, communications and computer sciences*, vol. 91, no. 1, pp. 211–220, 2008.

[18] E. K. De Jong, "Permutation of opcode values for application program obfuscation," Aug. 19 2008, US Patent 7,415,618.