On the Security of Safety-critical Embedded Systems: Who Watches the Watchers? Who Reprograms the Watchers?

Carlos Moreno and Sebastian Fischmeister
Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, Ontario, Canada
{*cmoreno, sfischme*}*@uwaterloo.ca*

Keywords: Embedded Systems Security, Physical Unclonable Functions, Safety-critical Systems, Runtime Monitoring.

Abstract: The increased level of connectivity makes security an essential aspect to ensure that safety-critical embedded systems deliver the level of safety for which they were designed. However, embedded systems designers face unique technological and economics challenges when incorporating security into their products. In this paper, we focus on two of these challenges unique to embedded systems, and propose novel approaches to address them. We first deal with the difficulties in successfully implementing runtime monitoring to ensure correctness in the presence of security threats. We highlight the necessity to implement runtime monitors as physically isolated subsystems, preferably with no (direct) connectivity, and we propose the use of program tracing through power consumption to this end. A second critical aspect is that of remote firmware upgrades: this is an essential mechanism to ensure the continuing security of a system, yet the mechanism itself can introduce severe security vulnerabilities. We propose a novel approach to ensure secure remote upgrades and sketch the details of an eventual implementation. It is our goal and hope that the computer security and embedded systems communities will discuss and evaluate the ideas that we present in this paper, to assess their effectiveness and applicability in practice.

1 INTRODUCTION

Security is an essential component in safety-critical embedded systems to ensure the required level of safety given the ever-increasing level of connectivity of these systems. The Internet of Things (IoT) and the prospect of connected vehicles makes the embedded devices an equally or more attractive target for cybercriminals, compared to the world of PCs. Research studies such as the Jeep Cherokee hack by Miller and Valasek (Miller and Valasek, 2015; Miller and Valasek, 2016), the pacemaker hack by Barnaby Jack (Computerworld Magazine, 2012), and real-world incidents such as Stuxnet in the nuclear domain (Langner, 2011) demonstrate the realistic nature of this threat. The increasing complexity and more sophisticated functionality of modern systems (including vehicles, medical devices, industrial control systems, etc.) only makes the task of adding security more challenging.

Correctness and reliability are also essential in safety-critical systems (Avizienis et al., 2004). These systems in general require runtime monitoring subsystems to enforce these aspects during the devices' operation phase (Pnueli and Zacks, 2006; Havelund, 2008; Navabpour et al., 2013). Indeed, regulatory standards mandate runtime monitoring at least for high criticality systems such as aerospace (RTCA / EUROCAE, 2012) and nuclear facilities (IEC, 2006); in the automotive field, ISO-26262 lists runtime monitoring as "highly recommended" for compliance with high safety integrity levels (ISO, 2011).

Adapting these ideas to the embedded systems security context poses unique and important difficulties: any efforts to ensure correctness and reliability, including fault-tolerance and redundancy mechanisms as well as runtime monitoring are in vain if a malicious attacker has the ability to reprogram the system or some key subsystems. Moreover, any monitoring specifically targeting security (i.e., the integrity of the system as it operates) is also susceptible to the threat of reprogramming the firmware of the device if it runs in the same hardware as the system being monitored. This was effectively demonstrated by the Jeep Cherokee hack (Miller and Valasek, 2015).

Problem Statement

Two key challenges are closely related to the above aspect, and constitute the motivating problem statement for this work:

Moreno, C. and Fischmeister, S.
On the Security of Safety-critical Embedded Systems: Who Watches the Watchers? Who Reprograms the Watchers?.
DOI: 10.5220/0006228304930498
In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP 2017)*, pages 493-498
ISBN: 978-989-758-209-7

- **How do we effectively monitor the integrity of the operation of an embedded device?** In particular, how do we incorporate runtime monitoring techniques that are virtually immune to the threat of remote attackers, including reprogramming firmware?
- **How do we securely perform remote firmware upgrades?** One key aspect is that ensuring security during the operation of a system requires having the ability to (remotely) reprogram it, so that security vulnerabilities can be corrected promptly after being discovered; thus, it is necessary to ensure that such mechanism cannot be abused by attackers to reprogram devices with malware of their choice.

Our Contributions

This paper addresses the two challenges highlighted above, and proposes novel and practical approaches to solve these problems. For the runtime monitoring aspect, we highlight the necessity of designing the security runtime monitor as a physically isolated device; to this end, we survey techniques where power consumption is used for anomaly detection or program trace reconstruction and argue in favor of mechanisms based on this promising approach.

We also propose a novel mechanism for secure firmware upgrade based on Physical Unclonable Functions (PUF), and present a sketch of a potential implementation; the novelty in our scheme is centered around the use of a single-challenge PUF to generate an encryption key that only the legitimate user has access to. Given the high level of resilience against physical attacks that PUFs exhibit, we believe that our proposed mechanism has the potential to provide a substantially high level of security.

Organization of the Paper

The remaining of this paper is organized as follows: Section 2 provides background on PUFs. Section 3 presents our proposed techniques, including power-based runtime monitoring in Section 3.1 and our proposed mechanism for secure firmware upgrade in Section 3.2. Section 4 includes a brief discussion and future work, followed by some concluding remarks in Section 5.

2 BACKGROUND – PHYSICAL UNCLONABLE FUNCTIONS

Physical Unclonable Functions (PUFs) are a relatively new and promising primitive where functions physically unique to each device can be produced (Herder et al., 2014). This has interesting applications as mechanisms for devices to authenticate themselves or for generation of encryption keys.

The main benefit of PUFs is their increased resilience to reverse-engineering compared to tamper prevention or tamper-resistance techniques used to protect some embedded secret information such as cryptographic keys. This derives from the fact that the uniqueness of this generated information comes from the unique variations in the manufacturing process at the microscopic level. Techniques typically rely on intentional race conditions in the hardware where the factor that resolves these race conditions is the unique microscopic characteristics of each device given by random variations in the manufacturing process beyond anyone's control. These microscopic characteristics are virtually impossible to measure and clone, even with the most sophisticated and expensive equipment and state-of-the-art techniques.

PUFs are classified into two main types: single-challenge and multiple-challenge. A single-challenge PUF can be seen as a constant function — the hardware produces a fixed output, with the important characteristic that this fixed value is unique to each device. Single-challenge PUFs are suitable to produce cryptographic keys. Multiple-challenge PUFs in general involve hardware that interacts with external input signals, and are useful for device authentication. This work relies on single-challenge PUFs, so we omit any additional details on multiple-challenge PUFs; the interested reader can find more details in (Herder et al., 2014).

PUFs are in general combined with some form of error correction code, to ensure that the output from the PUF is consistent across accesses (Herder et al., 2014); this is necessary to compensate for the fact that the race conditions in some cases may be resolved by the noise, producing outputs that vary across multiple accesses. This is especially important for single-challenge PUFs with outputs used as a cryptographic key.

3 OUR PROPOSED TECHNIQUES

This section presents our proposed techniques, along with our arguments to support the ideas and sketches of possible implementation details.

3.1 Runtime Monitoring Through Power Consumption

We propose the idea that, in addition to the standard runtime monitoring techniques implemented in

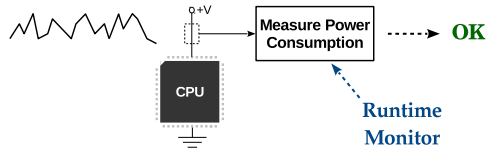
safety-critical systems to ensure correctness and reliability, these systems should include runtime monitors to enforce security properties, including the integrity of the software being executed. Ensuring integrity of the execution is essential, as several studies have demonstrated the ability of an attacker to reprogram the firmware of a device, bypassing any runtime monitoring subsystems that are running in the same processor as the standard functionality (Miller and Valasek, 2015; Computerworld Magazine, 2012).

Running the runtime monitors in the same processor as the standard functionality introduces important challenges in terms of risk of breaking extra-functional requirements such as timing or resource usage as well as ensuring isolation between the monitor and the standard functionality. Moreover, "system crashes" due to corruption in the execution environment (e.g., stack or memory corruption) can disrupt or entirely disable the runtime monitor. This constitutes an important advantage for runtime monitors that are physically isolated from the processor being monitored. This aspect is even more critical when we consider security, given the threat to the integrity of the software being executed.

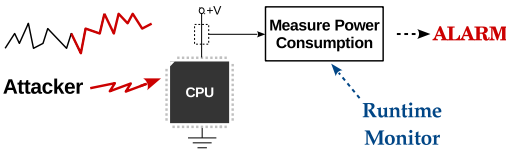
On the other hand, running in the same processor alongside the software being monitored has the advantage that more information is available to the runtime monitor, increasing its potential effectiveness. Thus, we posit the approach of using two runtime monitors: one that runs in the same processor as the software being monitored to enforce correctness and reliability; and one that runs as an isolated device to ensure security — possibly in addition to enforcing properties related to correctness and reliability. We observe that a system crash, regardless of whether it is related to a security attack, is detected by the external monitor.

It is also important in a practical implementation to ensure that the external monitor system is physically isolated from the Internet or in general from any form of remote connectivity. At the very least, a carefully designed *air gap* mechanism should be included, ensuring that the monitor lacks any capabilities of wireless connectivity and any firewall functionality implemented in FPGA or otherwise reprogrammable firmware.

We propose the use monitoring through power consumption, since this is a technique with a remarkable potential for effectiveness. Given the strict physical relationship between execution and power consumption, an attacker would have to "bend the Laws of Physics" to be able to inject malware that exhibits the same power consumption profile as the original software in the device. At the very least, the attacker would be limited to executing software that produces a power consumption profile "close enough" to the original (which they may or may not know, depending on the specific system), and yet do something useful for the attacker. Figure 1 illustrates this approach.



(a) Authentic Execution



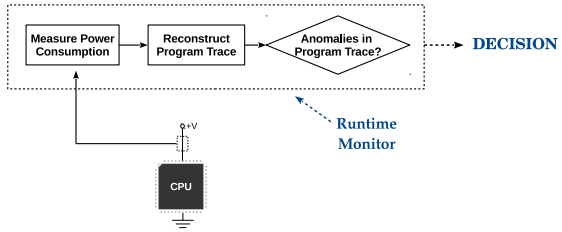
(b) Attacker Injects Malware

Figure 1: Runtime Monitoring Through Power Consumption.

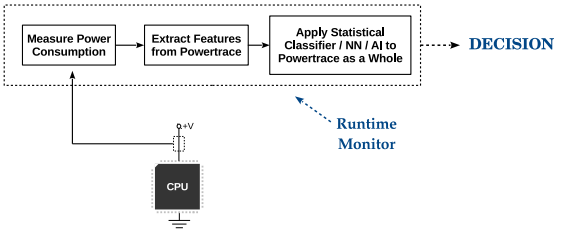
Two different approaches have recently appeared in the literature: the monitor system can reconstruct the execution trace and then analyze the program's execution directly, or it can analyze characteristics of the power trace and detect deviations from normal patterns. This is illustrated in Figure 2.

The former approach has been demonstrated in (Moreno and Fischmeister, 2016; Liu et al., 2016; Moreno et al., 2013; Msgna et al., 2013). These techniques can be combined with the work presented in (Moreno et al., 2016), where a compiler optimization stage is used to reorder instructions in the generated binary code to maximize distinguishability between the power traces for different sections of the program, potentially increasing the effectiveness of the program trace reconstruction.

Clark et al. (Clark et al., 2013) presented an approach based on applying pattern classification techniques to the power trace as a whole. This technique can be useful for devices with simple and highly repetitive tasks (as those presented in that work), and it may also be suitable for analysis of a power trace containing power consumption of multiple devices, where it may be easier than attempting to disaggregate the data in the power trace. Though the work by Clark et al. uses a statistical pattern recognition approach, this technique may be combined with neural networks or any other machine learning techniques, even if, to the best of our knowledge, none of these have been attempted by the research community.



(a) Reconstruct Execution Trace and Analyze



(b) Apply Classification Techniques to the Power Trace

Figure 2: Runtime Monitoring – Processing Approaches.

3.2 Secure Remote Firmware Upgrades

Though the standard practice of software upgrades with security fixes to patch vulnerabilities as they are found has been criticized, no convincing and practical alternatives have been proposed. The evidence pointing to the need of this practice at the very least as a "safety net" is compelling: examples such as Heartbleed (OpenSSL Team, 2014), Bash Shellshock (NIST, 2014), and POODLE (Möller et al., 2014) highlight the importance of upgrade mechanisms; the use of recognized tools that are assumed to be secure could have unexpected critical security flaws, potentially introducing severe vulnerabilities in our system. Indeed, (McAfee, 2015) describes this practice as essential to ensure security in automotive systems while reducing the cost of recalls. (SAE, 2016) mentions this aspect as a recovery mechanism that still requires more research to be securely applied and accepted by customers.

However, this practice is especially difficult given the unsupervised nature of embedded systems, making it hard to prevent abuse of this mechanism for the purpose of reprogramming the firmware with the attackers' malware. (McAfee, 2015) mentions "appropriate user controls and safety precautions", but it provides no concrete examples of any efficient and effective methods to accomplish it.

In low-criticality systems, where the required security level is moderate, the designer can embed a public-key in the device so that firmware upgrades can be digitally signed. The device verifies the signature and only applies the upgrade if the verification passes. However, this requires tamper-resistance mechanisms to protect the integrity of the embedded public-key, which may not provide a high enough security level for use in safety-critical systems.

We propose a secure remote firmware upgrade mechanism based on an alternative form of using single-challenge PUFs. We describe the operation of this mechanism and provide a sketch of possible implementation details. Our goal and hope is that the communities of computer security and embedded systems (including industry) will discuss, evaluate and critique our proposed technique, eventually leading to practical implementations.

The approach is based on using the output of a single-challenge PUF as the cryptographic key that will be used to encrypt the binary with the new firmware that the device receives. However, using encryption keys generated by PUFs leads us to a tricky situation: if the device never outputs this key, then it is only useful to encrypt data at rest (i.e., data encrypted by the device that only the device itself will read and decrypt at a later time). If we want to use this PUF to generate a key to communicate with the device in a way that the device can rely on the authenticity of the source, then the situation gets more complicated: the (legitimate) user has to possess this key so that they can encrypt the data transmitted to the device. Thus, a mechanism for the user to request the key from the device is necessary. This introduces a problem: if the user can request the device to output the key, then an attacker can also request and obtain the key. We observe that encrypting that very transmission (the device transmitting the key) or attempting to authenticate ourselves to the device, so that it only outputs the key to the legitimate user, are not effective approaches: to be able to accomplish that, we need an additional shared secret with the device; that is, we need another secret key embedded in the device. However, the premise is that PUFs provide a superior mechanism to hide secrets in a device, so an additional auxiliary secret would simply introduce a weaker link, making the use of the PUF pointless.

With ECC (or in general with public-key cryptography), we could get around this limitation, but only in one direction and would not solve the above problem: the device could use the output of the PUF as the private key, and generate the corresponding public key (examples and details can be found in (Hankerson et al., 2004)), which can be disclosed without any restrictions. The problem then is that anyone can use this public key to encrypt transmissions to the device, and the device has no way to authenticate the sender of those transmissions (again, unless

some other shared secret is available, which as mentioned is not a valid option).

The key observation is that the legitimate user of the devices has one important advantage: it can be the first party to ever request the device to disclose the key (the output of the PUF). In our proposed mechanism, the device incorporates a protocol for one-time disclosure of this key, with this disclosure being unconditionally followed by a step that *physically* severs the path from the PUF to device bus or external pins. Thus, we can safely use this key to communicate with the device, since it is virtually guaranteed that no-one else will be able to obtain the key. Figure 3 illustrates this idea.

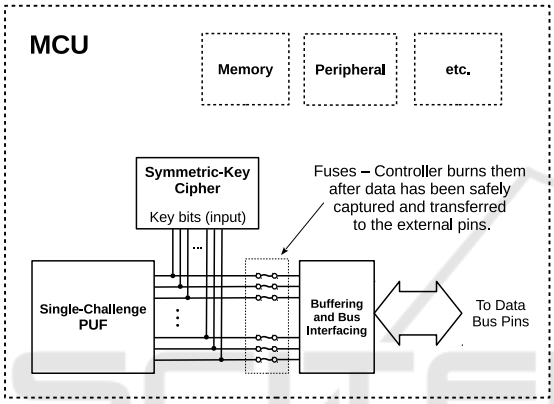


Figure 3: Mechanism for One-Time Disclosure of Encryption Key.

We notice that an insecure alternatively would be for the device to include an externally issued command that severs the path between the PUF and the outside of the device. This certainly provides more flexibility, but it introduces two risks: (1) the user could fail to issue the command to sever the path from the PUF to the outside, leaving the key accessible without any restriction; and much more severe, (2) a malicious manufacturer of the chips with this functionality (or a manufacturer with its infrastructure compromised) or a malicious entity in the supply chain could steal and keep the keys before shipping the chips to the legitimate user. We need the device-enforced aspect of self-destruction upon first access, so that if someone steals the key, the legitimate user will detect it and they will simply discard the device.

The particular scenario of interest to us is that where the output of a PUF is used as the encryption key for a firmware upgrade. The vendor of the devices in this case extracts the key for each device before shipping them and *securely* stores them in a database. These keys can be later used to encrypt the transmitted binaries with the new firmware. This mechanism prevents anyone other than the manufacturer from replacing the firmware of those devices. The guarantee is quite strong on the side of the device, since it has the same advantages as PUFs in general. If the mechanism to physically sever the path from the PUF to the outside world is properly implemented, then we can have a strong assurance that no-one else will be able to extract the key from the device. One point of failure does remain, of course: the keys are stored in a database under the manufacturer's control, which may be subject to attacks. However, with the legitimate operator of the system being in physical possession of that database, strong layers of security can (and must, of course) be added.

Multiple instances of this mechanism (i.e., multiple PUFs with one-time output disclosure) could be included for various purposes; for example, an additional encryption key could be used for regular communications where the device needs to authenticate incoming transmissions. Though the same key generated by the one PUF in the device could be used for multiple purposes, using separate keys for different purposes provides an increased level of security, so we suggest that at least the key used for encrypting the binaries in firmware upgrades should be separate from any keys for other purposes.

4 DISCUSSION AND FUTURE WORK

Some important challenges need to be solved for our proposals to be feasible approaches in practice. Though we plan to tackle some of these issues, we hope that the communities of computer security and embedded systems will also take on these challenges.

For the power-based runtime monitoring, the aspect of an adequate action by the system upon detection of anomalous behavior is crucial. We believe that this is rather an engineering aspect, and it is specific to the particular system and application domain where the technique is used. Aspects such as real-time operation given reasonable amount of computing power and minimization of training database size are also important. Information from additional channels could further improve performance by providing clues about the current operation of the program. An example of this is the use of timing of communications in the CAN bus (Cho and Shin, 2016).

Our proposed remote firmware upgrade mechanism also has important engineering aspects that need to be solved, such as storage and management of the devices' keys, and protocols for the actual upgrades, which require generation of unique encrypted binaries for each device. On the research side, the communi-

ties still need to investigate the feasibility and effectiveness of the method, eventually leading to practical implementations. Regarding the use of one-time-disclosed PUF-generated cryptographic keys for multiple applications, we emphasize our recommendation of the use of separate keys for different purposes, or at the very least, one separate encryption key for the purpose of firmware upgrades.

5 CONCLUSIONS

In this paper, we have proposed and discussed techniques for runtime monitoring of security properties in safety-critical embedded systems and for secure remote firmware upgrades. These techniques are aimed at solving two related problems that become critical as the level of connectivity of these systems increases. Some important challenges remain to be solved before these techniques can be effectively applied in practical systems, and it is our hope that the communities of computer security and embedded systems will evaluate and discuss these techniques, eventually leading to practical implementations.

ACKNOWLEDGEMENTS

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Ontario Research Fund.

REFERENCES

Avizienis et al. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

Cho, K.-T. and Shin, K. G. (2016). Fingerprinting Electronic Control Units for Vehicle Intrusion Detection. *USENIX Security Symposium*.

Clark et al. (2013). WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices. *USENIX Workshop on Health Information Technologies*.

Computerworld Magazine (2012). Pacemaker Hack Can Deliver Deadly 830-Volt Jolt.

Hankerson, D., Menezes, A., and Vanstone, S. (2004). *Guide to Elliptic Curve Cryptography*. Springer-Verlag.

Havelund, K. (2008). Runtime Verification of C Programs. In *International Conference on Testing of Software and Communicating Systems*.

Herder, C., Yu, M.-D., Koushanfar, F., and Devadas, S. (2014). Physical Unclonable Functions and Applications: A Tutorial. *Proceedings of the IEEE*, 102(8).

International Electrotechnical Commission (2006). Nuclear Power Plants – Instrumentation and Control Systems Important to Safety – Software Aspects for Computer-Based Systems Performing Category A Functions (IEC-60880).

International Office for Standardization (2011). International Standard ISO-26262 – Road Vehicles Functional Safety.

Langner, R. (2011). Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy*.

Liu et al. (2016). On Code Execution Tracking via Power Side-Channel. In *ACM Conference on Computer and Communications Security*. ACM.

McAfee (2015). Automotive security best practices.

Miller, C. and Valasek, C. (2015). Remote Exploitation of an Unaltered Passenger Vehicle.

Miller, C. and Valasek, C. (2016). Advanced CAN Injection Techniques for Vehicle Networks.

Möller, B., Duong, T., and Kotowicz, K. (2014). This POODLE Bites: Exploiting The SSL 3.0 Fallback – Security Advisory.

Moreno, C. and Fischmeister, S. (2016). Non-intrusive Runtime Monitoring Through Power Consumption: A Signals and System Analysis Approach to Reconstruct the Trace. *International Conference on Runtime Verification (RV'16)*.

Moreno, C., Fischmeister, S., and Hasan, M. A. (2013). Non-intrusive Program Tracing and Debugging of Deployed Embedded Systems Through Side-Channel Analysis. *Conference on Languages, Compilers and Tools for Embedded Systems*.

Moreno, C., Kauffman, S., and Fischmeister, S. (2016). Efficient Program Tracing and Monitoring Through Power Consumption – With A Little Help From The Compiler. In *Design, Automation, and Test in Europe (DATE)*.

Msgna, M., Markantonakis, K., and Mayes, K. (2013). The B-side of side channel leakage: control flow security in embedded systems. In *International Conference on Security and Privacy in Communication Systems*, pages 288–304. Springer.

Navabpour et al. (2013). RiTHM: A Tool for Enabling Time-triggered Runtime Verification for C Programs. In *Foundations of Software Engineering*. ACM.

National Institute for Standards in Technology (2014). BASH Shellshock – CVE-2014-6271 (Exported function through environment variable).

OpenSSL Team (2014). OpenSSL Heartbeat Read Overrun – CVE-2014-0160.

Pnueli, A. and Zacks, A. (2006). PSL Model Checking and Run-Time Verification via Testers. *International Symposium on Formal Methods*.

RTCA / EUROCAE (2012). DO-178C: Software Considerations in Airborne Systems and Equipment Certification.

SAE International (2016). Cybersecurity Guidebook for Cyber-Physical Vehicle Systems.