Binding Hardware and Software to Prevent Firmware Modification and Device Counterfeiting

Robert P. Lee
robert.lee.2013@live.rhul.ac.uk
Konstantinos Markantonakis
k.markantonakis@rhul.ac.uk
Raja Naeem Akram
r.n.akram@rhul.ac.uk

Smart Card Centre, Information Security Group, Royal Holloway, University of London
Egham, Surrey, TW20 0EX, United Kingdom

ABSTRACT

Embedded systems are small scale computing devices that are increasingly located in more of the items we use and own. The number of embedded systems in the world is increasing dramatically as the "internet of things" concept becomes more prevalent in the market. The value of the market for embedded systems is predicted to increase to being worth trillions of dollars by 2020. With great value in the embedded system market, there is a need for preventing unauthorised firmware tampering or product counterfeiting. Here is presented a technique for binding software to hardware instances that uses hardware intrinsic security properties of the devices being protected. The proposed technique provides assurance to manufacturers that only they can perform their hardware and software binding and create their products. Also presented is an FPGA implementation of the described scheme that binds the hardware and software together with only a 6.7% increase in execution time. Thus, making it difficult for an attacker to either counterfeit the device or extract the (software) Intellectual Property.

Keywords

Binding; hardware; software; intrinsic; security; PUF; firmware; modification; counterfeiting.

1. INTRODUCTION

Embedded systems are used in many different settings such as in mobile phones, MP3 players, cars, aeroplanes as well as fridges and utility meters [15]. With the rise of ubiquitous computing and the "internet of things" (IoT), it is becoming more common for computer systems to be included in more devices [19]. The IoT concept has gathered interest from businesses with some estimating the market to be worth trillions of dollars by 2020 [22]. However the market for counterfeit electronic goods is also increasing with some suggesting that 10% of all IT products sold are counterfeit [1]. A further risk to products comes from the nature of their manufacturing. In some areas, such as graphics cards, it is common to produce just a single design of integrated circuit (IC) and then construct different level products by including or excluding cores or features using the device firmware. Vendors who develop products in this manner are at risk of their sales revenue affected by people who buy cheaper products and load them with software from more powerful devices in order to unlock premium features or performance [21]. With large amounts of money available to be made or lost it is important for organisations to be able to prevent unauthorised firmware modification and platform counterfeiting.

This paper considers the problem of securely binding software to individual hardware devices. If a piece of software has been bound to a hardware device then this ensures that the software will only run on the hardware it has been loaded onto. We propose that binding a piece of software to a hardware instance can be used to prevent unauthorised firmware tampering, counterfeiting of "platforms" or potentially be used to protect the intellectual property of a manufacturer. In this paper, a "platform" is a deployed product considered as an entire entity comprised of both hardware and software.

The intuition behind this paper is that unauthorised firmware alteration and counterfeiting can be prevented in the following way. Only the legitimate manufacturer is able to install or provision software for execution on their platforms because only they are able to bind software to their hardware. Conversely, attackers are unable to use alternative firmware or create counterfeit products using legitimate products because software extracted from a legitimate product cannot be copied onto an illegitimate hardware as it will only operate correctly on the hardware it was bound to.

We propose the following setting as a real-world case study for our problem. A manufacturer has developed a Graphics Processing Unit (GPU) for use in graphics cards. They send the design of the GPU to be fabricated and after receiving the manufactured chips run tests on the ICs. The GPUs are then tested to see how successfully they have been fabricated. After testing the GPUs are split into three sets: the chips that performed best, those that performed worst and also the chips that were of middling standard. The three groups of chips are all included into graphics card which are loaded with firmware that determines the voltage and clock speed the GPUs are to be set to. The performance settings included in the firmware are assigned based on what the IC tests show they are capable of.

One threat to the manufacturer in the case study described previously is that some may wish to have a top-level product

ACM ISBN .
DOI:

without paying the full price for it. Those wishing to do this may purchase a mid-level product and then replace the firmware running on the graphics card purchased. If the IC is made to the same design on all devices then it may be that a middle level product is able to run at the same clock speed and with all the features of a top-level product. This may be the case if the IC testing is overly cautious in order to ensure that the products sold are as reliable as possible. In that case, switching the firmware on a mid-level product for that of a top-level product could allow customers to avoid paying full price for the best graphics card. The business of the manufacturer will suffer if there are many customers who choose to avoid buying the top-level product in favour of buying the cheaper products and modifying them.

However, if the graphics card firmware were securely bound to the devices it is installed onto then this attack would be prevented. Firmware taken from a top-level product would have been bound to the device it was taken from and would fail to execute correctly on any other device it was loaded on. Therefore, customers who wished to own a device with the same performance and features as the top-level product would be required to buy the top-level product.

In this paper we make the following contributions:

- This paper identifies a new problem setting for binding software with hardware devices which considers a powerful attacker who can access all device storage as well as duplicate hardware created from the same specification as the original (Section 2).
- We propose a flexible technique which would allow an application to be bound to a hardware instance and thus prevent unauthorised software from executing on the hardware device (Section 4).
- We describe several different security primitives which could be used in the proposed scheme in order to securely bind a software program to a hardware instance (Section 4.3).
- We have also developed a prototype implementation of the proposed scheme that demonstrates how it may be used and at what performance cost (Section 5).

The paper is structured as follows: Section 2 contains a more detailed examination of the problem considered in this paper. Section 3 contains a survey of previous work related to this paper. Section 4 contains the solution we are proposing to the problem considered in this paper. Section 5 describes the proof-of-concept implementation of the proposed solution that has been developed. Section 6 concludes the paper and describes the future work still required in this area.

2. PROBLEM DESCRIPTION

In this section we will expand on the previously described settings in order to formalise the problem we are considering.

2.1 Notation

The following notation will be used when describing the problem.

A	The attacker who is attempting to create counterfeit platforms.
x, y	Platforms comprised of a hardware and software element.
$SW(x)$	The software which is personalised to a platform x.
$HW(x)$	The hardware of the platform x.
$MEM(x)$	The memory of a platform x. This includes both the *persistent* and *non-persistent* storage of the platform x.

2.2 Problem Scenarios

The different scenarios considered in this paper are listed here.

1. The manufacturer of a platform has a hardware device $HW(x)$ and software which will be run upon it. The manufacturer wishes to personalise the software for use on only the hardware device $HW(x)$, creating a device-personalised software $SW(x)$, which is loaded onto the device $HW(x)$. This process creates the platform x which is the amalgamation of $HW(x)$ and $SW(x)$.
2. The platform manufacturer wishes to ensure that only they are able to install software onto the products they create. The manufacturer is seeking to prevent the owner of a legitimate device $HW(x)$ from executing alternative software to the $SW(x)$ which was originally loaded onto the platform x.
3. The manufacturer of a platform wants to ensure that there does not exist an attacker A who can create counterfeit versions of his platforms. We assume that the attacker will seek to create a counterfeit platform y by purchasing a legitimate platform x, extracting the contents of $MEM(x)$ and loading them onto an illegitimate device $HW(y)$ where $HW(y)$ has been built to the same specification as $HW(x)$. How the attacker A is able to access hardware such as $HW(y)$ is outside of the scope of this paper, however it may be that $HW(y)$ is a device in the same product line as x but a different level of product as in the case study described in Section 1.

2.3 Attacker Model

The attacker A shall have the power to:

1. Read and copy the entire memory, $MEM(x)$, of the device.
2. Make use of software, $SW(y)$ taken from legitimate devices. The software the attacker can access is software extracted from legitimate devices and will be bound to the devices it has been taken from.
3. Make use of hardware built to the same specification as the device it wishes to counterfeit. The attacker can create a hardware for a counterfeit platform y such that $HW(y)$ is created from the identical specification as $HW(x)$ for a target platform x. Note that our attacker does not have the ability to exactly copy any properties intrinsic to the original hardware $HW(x)$, only to create hardware according to the same design. Using properties intrinsic to hardware is described in Section 3.2.
4. Read and copy any data which is loaded onto any of the buses which make up the embedded system.

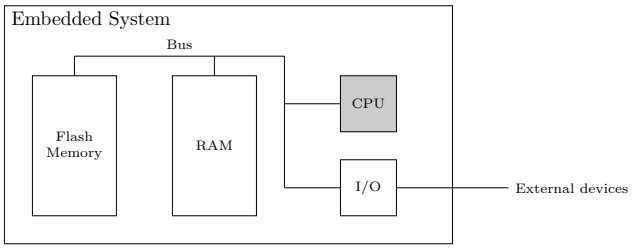

Figure 1: Block diagram of a generic embedded system's hardware

The attacker A has achieved their aim if they achieve either of the following:

1. The attacker creates a functioning, duplicate platform y which cannot be differentiated from a legitimate platform x.

2. The attacker installs a software $SW'(x)$ which can successfully execute on a hardware $HW(x)$, where $SW' \neq SW$. The software SW is the software which was loaded onto the device by the manufacturer.

2.4 Assumptions

It is assumed that the application is suitably complex that it would be non-trivial for an attacker to simply recreate it. If the application was simple, then the most practical approach for an attacker would be to write their own version. It follows that developing a product equivalent to the original is likely to require a significant investment from the attacker; this would make their counterfeiting venture significantly less profitable.

We assume that the attacker is unable to read the contents of any registers which are part of CPU. The attacker is also unable to directly query any individual parts of the CPU. The attacker is able to access any of the memory of the device and observe data sent via buses, however the attacker is not able to see within the processor itself. A diagram of the embedded system being considered in this paper, and the elements of it accessible by the attacker, can be found in Figure 1; the grey portion indicates the "safe zone" that cannot be viewed or accessed by the attacker.

We also assume that the manufacturing of platforms is carried out in a secure location. The attacker is not able to choose, access or interfere with the software before it is loaded onto a platform. The attacker is only able to access software which has been bound to a platform by the manufacturer.

2.5 Design Requirements

To prevent the attacker from achieving its goals, a solution to the problem posed in this paper must meet the following requirements.

1. **The proposed technique must grant the ability to securely personalise software to a hardware device.** The security of the personalisation shall ensure that only the legitimate manufacturer is able to personalise their software. If only the manufacturer is able to personalise their software, this technique assures that only they are able to provision software for their products. The technique proposed must account for the fact that tamper resistance storage, that would allow us to securely store/have keys, is not available.

2. **The software personalisation must make use of a hardware fingerprinting technique.** This proposed technique must allow the software running on the hardware device to be bound to the specific hardware instance that it is personalised for. Personalising the hardware to a property intrinsic to the device would provide confidence that software on one device cannot be copied onto another. A property intrinsic to the hardware instance is required because it is assumed that tamper-resistant storage is not available.

3. **The solution must protect all data that is to be stored in any of the storage areas that are present on the device.** This requirement is necessary because the attacker is able to access any of the memory contents of the platform. If the data is ever unprotected in the memory of the device then an attacker could gain data unbound to the platform. Because the attacker can access all device storage, the only "safe zone" it cannot view or access is within the processor of the device.

3. RELATED WORK

This section describes related work in the areas of preventing counterfeiting, unauthorised device modification and hardware intrinsic security. The first half of this section considers different anti-counterfeiting and device integrity protection techniques in the context of the problem described in Section 2. The second half considers the use of physically unclonable functions for hardware fingerprinting.

3.1 Anti-counterfeiting measures

The piracy and protection of Intellectual Property (IP) is an area which has received a large amount of interest as will be discussed below.

One technique for preventing device counterfeiting is to use Trusted Platform Modules (TPMs) to ensure the integrity of applications, securely store cryptographic keys or provide a secure boot process [20]. The cryptographic operations and storage of a TPM are attractive because they exist in a tamper resistant environment which protects data and prevents counterfeiting [26]. However, TPMs are not suitable for solving the addressed problem which assumes that there is no tamper-resistant storage; in this setting trust cannot be placed in the security offered by a TPM.

A software-based method for protecting IP is making use of code obfuscation. Simply put, code obfuscation is the act of "jumbling up" a program in order to render a program very difficult to analyse/modify while retaining performance [2, 5, 25].

Code obfuscation could be applied to solving the problem posed in this paper as it would prevent the application data from being understood by an attacker. However, preventing attempts to analyse the program would not be sufficient for our requirements. Our attacker has the ability to copy all data from one platform and load it onto another. An obfuscated program copied from a different, legitimate device would operate on the device it was loaded onto as it did on the device it was taken from. Code obfuscation is designed to protect a company's IP by preventing reverse engineering

or third party modification of the firmware rather than unauthorised replacement of firmware. The difference between the goal of obfuscation techniques and the problem in this paper leads to the conclusion that an obfuscation technique would not be suitable for our purposes. Another concern with using obfuscation is that there are some who claim that secure obfuscation is not possible [3, 4].

Unlike other solutions described in this section, our problem is one of controlling platform manufacturing in order to prevent unauthorised firmware modification and product counterfeiting. This problem was considered by Kean who, in 2002, introduced the problem of a company providing licenses for a limited number of uses of a Field Programmable Gate Array (FPGA) IP core to a customer [13]. Kean's techniques allow for a different licensing model by moving from a large up-front fee to licensing per FPGA IP core. The protocol contained two main steps: providing the bitstream to the customer and loading the IP core onto the FPGA. To prepare the bitstream it would be encrypted by the company providing the IP core and then it would be sent to the customer. A trusted programming software would be used by the customer to decrypt the bitstream and then customise the bitstream using secrets known only by itself and the FPGA. The customised bitstream would then be loaded to the FPGA [13].

However, this solution is not suitable to this work because while Kean's scenario is related to that considered here, it is different. Firstly, Kean considers an IP producer selling uses of an FPGA bitstream to customers whereas in this work the entity programming the platform is the developer of the IP and not a separate entity. Secondly, this work is not interested in limiting how many times a legitimate IP may be used, a solution is required to prevent any unauthorised modification of device firmware.

3.2 Hardware Intrinsic Security

Hardware intrinsic security is a field which encompasses several different areas concerned with trying to use or protecting against some of the physical properties of hardware devices such as side channel leakage or random variations in hardware fabrication.

3.2.1 Physically Unclonable Functions

Physically Unclonable Functions (PUFs) were first introduced in 2002 by Gassend et al. [8]. A PUF can be considered as a fingerprint for an electronic device.

A PUF is a circuit which if two copies are constructed, then they will not have precisely the same behaviour. PUFs have been the subject of a large amount of study and many different designs of PUFs have been proposed [10, 16, 18, 24]. The security (or lack thereof) of PUFs is a question without a strong answer however some circuits have been proved to be much more effective PUFs than others [12].

PUFs have been used as part of solutions to various security problems. One area that PUFs have been used in is as part of authentication mechanisms [6, 7].

The potential for using PUFs for preventing counterfeiting was first described by Simpson and Schaumont in 2006. In their paper they described a technique by which hardware and software developers could combine their products with confidence that their IP is secured. PUF challenges and responses for the hardware device are stored with a trusted third party (TTP); using these and encryption, the hardware and software can authenticate with each other offline [23]. The work of Simpson and Schaumont was extended in 2007 by Guajardo et al., who simplified original protocol and removed the need for secure channels, preventing even the TTP from seeing the IP of the software developer [10].

The solutions produced by Simpson and Schaumont, and the later improvement by Guajardo et al., controls the ability for an IP to be used on a piece of hardware. However their problem scenario is significantly different from that of this paper which results in their solution being inapplicable to our problem. The authors are solving a problem wherein a system developer (SYS) wishes to use the IP of an IP Provider (IPP). These parties use a TTP to authenticate in order to ensure that the genuine IP is loaded onto a genuine SYS's product. However, in our problem the SYS and IPP are the same company so will not require a TTP for communication.

PUFs have also been applied to problems similar to that of this paper of protecting IP, as by Guajardo et al. in 2008 or by controlling the ability to manufacture a platform, as by Gora et al. in 2009 with later work by van der Leest et al. in 2012 [9, 11, 27]. The paper of Gora et al. introduced the problem of binding a piece of Software Intellectual Property (SWIP) to a particular FPGA in order to ensure that the SWIP would function on the intended FPGA only [9]. Their technique consists of two main components: an enrolment phase and an operational phase. Firstly the enrolment consists of extracting the encryption key to be used from the PUF on the FPGA device and using it to encrypt the SWIP. The operational phase is where the device has been deployed and is used and begins with a secure boot process, during which the SWIP is decrypted by the device. After decryption the SWIP is used by the FPGA which can now execute the SWIP as normal [9].

The proposal by Gora et al. is a solution which effectively meets several of the requirements defined in Section 2.5. A SWIP is successfully bound to a HW device and the technique takes steps to protect the SWIP while it is stored on the device. However, there are some areas for improvement on their scheme. Firstly, the attacker described in Section 2.3 can access any part of the storage on a platform and there are no restrictions on when the attacker can use its access to the storage. Against the scheme of Gora et al. our attacker could wait until the secure boot has decrypted the SWIP before copying it from the storage. However, in the proposed scheme there the unmasked application is only ever present within the processor and is never stored in the memory.

Another approach to including PUFs in processor execution was published by Kleber et al. in 2015 [14]. Their paper presents a Secure Execution PUF-based Processor (SEPP) that uses a PUF generated key with AES in counter-mode to encrypt the instructions stored in the devices memory. The SEPP may meet the requirements of a solution for the problem of this paper, however the performance impact of using their technique is large. Therefore a more specialised solution to the problem is required in order to preserve performance while protecting against attackers.

4. PROPOSED SOLUTION

In this section we provide a description of the scheme proposed as a solution for the problem described in Section 2.

4.1 Design Overview

We consider the case that the software ran on a platform x

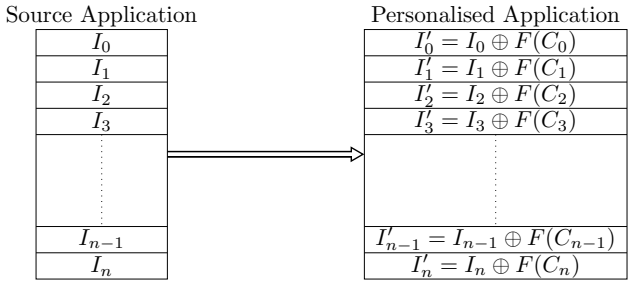

Figure 2: The transformation of source memory content into a personalised application

is comprised of a set of instructions I_i, where i is the memory location that the individual instruction I_i is stored in. We assume that the instructions comprising the program are stored in memory sequentially without gaps. Different portions of code may be navigated to via control flow instructions (such as `jump` or `call` instructions). However the code is stored in one unbroken portion of memory.

As stated in Section 2.5, we wish to personalise an application for a device. This requirement will be met by using a masking technique to conceal the instructions/data wherever they are stored on the device.

To meet with the listed requirements (Section 2.5) we require the masking to be specific to the device in some way. We will mask each instruction by using a challenge/response function F. F will yield different masks for instructions/data based on whatever the challenge for that data is. The use of F must also make use of some property intrinsic to the device. The masked instructions will be written as I'_i, and will be calculated by the following function:

$$I'_i = I_i \oplus F(C_i).$$

Where C_i is the challenge used for calculating the mask for the instruction I_i. However, requiring challenges for each instruction adds two additional problems: how much storage will be required for the challenges and how will the challenges be chosen. To address these problems we propose the following method for assigning the challenge values:

$$C_i = I'_{i-1}.$$

This choice of C_i reduces the storage requirement for the challenges used in masking the application instructions to one single challenge. The one challenge required is the C_0 that would be generate by a secure random number generator and would be used in protecting the instruction in the first location in the memory. A diagram of the transformation applied to the contents of memory when personalising for a device is included in Figure 2.

An alternative approach would be to use the memory address as the challenge value, however this would introduce a security flaw in a multi-application setting. In a multi-application setting, each application will be loaded from virtual memory address 0 onwards. If applications are produced by different developers then each application could be vulnerable to attacks from the developers of the other applications. The attack would involve a platform x which contains applications from two developers A and B. Developer A would buy a legitimate product which contains both applications, A would then copy their own application from the platform. Using the source code for its application, A could recover the mask values used to conceal his application; the responses to $F(0)$–$F(n)$ where n is the length of the application as stored on the platform x. Developer A could then recover the first n instructions of the application developed by B as it would have used the same challenges and responses. However, the proposed choice of challenge avoids the possibility of this type of attack.

4.2 Securing Mutable Data

The ability to secure any of the data which must be stored on the platform, but which may change, is a significant challenge for solutions to our problem.

Unlike when protecting the application instructions, cascading previous masked values as challenges is not practical for data that may change. If the contents of one memory location were changed, the subsequent value would have to be updated to reflect the change in its challenge value. After the immediately following value was updated, the next value would also have to be updated to reflect the different challenge stored. Updating the values in memory would continue until the end of the memory used by the application was reached. The large number of updates would add a considerable overhead to the storing of data in memory.

One potential solution to this problem is to store an extra challenge value for each mutable value. This would result in an increased storage requirement for platforms using our defence mechanism. However it would prevent one change of a stored value requiring updates to all the following stored values, as was previously described. With the challenges needing to be stored, as well as the data, it is likely that they will require regular updating to prevent data leakage. This approach would incur a large storage overhead as the amount of data to be stored would be doubled. Alternatively, the mutable data could be split into smaller groups of values that would each be allocated an extra challenge value. This technique would limit the amount of extra challenges required however if a value is updated then the proceeding locations in the same group would need updating. This approach would allow a tradeoff to be made between performance and memory overheads.

One remaining open question surrounding memory is how to handle memory that is requested/allocated at runtime. A similar technique to protecting other mutable data could be used to protect the data stored in allocated memory. Therefore, extra challenges would also be required for each dynamically allocated memory location. Any extra challenges required for dynamically allocated memory would need to be generated and used securely to avoid giving away information to attackers. Care will also be needed to prevent applications accessing the memory locations that are used to store the extra challenge values.

4.3 Choice of Function F

The central element in the technique which we propose is the use of a function which creates a masking value based on an inputted challenge. The security of this element is critical to the security of the entire technique and so it must be chosen with care. It is required that the function used for F must be able to provide a value of output based on an input challenge.

In the setting we propose for the use of the function, F. We require that the output of F must be unpredictable because if the output of F could be predicted from the input, an attacker could recover the data stored. Therefore, as the challenges are known to an attacker, if the function used is public, it must also make use of something secret. If a closed function were chosen for F, the unpredictability would stem from the unknown nature of the function. However this would not be advised by the authors as it would result in the platform only being protected by a "security through obscurity" approach.

We propose that there are several different potential candidates which may be chosen for F. For example, hash functions, block ciphers or PUFs could all perform as required for F. However, as each of these different options have significantly different properties they would each need to be used in different ways. These different uses would be needed in order to ensure the security of the proposed binding technique. Due to the different properties of the candidates for F, different amounts of extra information will be needed for each. The changes in required information will also have an impact on what secrets are required for each candidate function. Note that we also assume that only public functions will be used for F and we assume that attackers will know our choice of F.

We are now going to consider how the suggested functions may be used for F. One potential type of function which could be used for F are hash functions. Using a hash function does not address all of our requirements as it does not incorporate any properties intrinsic to the hardware. If a hash function were to be used for F then an extra element would need to be added in order to use hardware intrinsicness in protecting the application. This extra element could be a value representing a hardware fingerprint of the device which could be mixed with each challenge before hashing. A PUF could be used to provide the secret, hardware fingerprint value which would be mixed with the input or output of the hash function.

Block ciphers could also be used for the function F. A secure block cipher would ensure that only an entity who knows the secret key would be able to calculate the mask values used to conceal the application. However, similarly to using a hash function simply, a block cipher alone will not meet our requirements as there is no use of hardware intrinsic properties. A simple method for incorporating hardware intrinsicness could be to use a PUF to provide the key used by the block cipher. This would provide assurance that duplicate hardware constructed from the same specification would not possess the key used in binding an application to a different device.

A drawback to using a block cipher or hash function for the function F is that it would result in a large number of encryptions/hashes being performed. If each instruction were masked separately then a performance penalty may be incurred with methods using these types of functions so grouping instructions into basic blocks may be considered in order to require less encryptions/hashes to be computed.

A third type of function which could be used for F is a PUF. PUFs have many properties which would make them useful in solving the problem posed in this paper. Firstly they are an element of hardware which, by definition, will be unique[1] per device. If a PUF is included in the platform then it will ensure that *exactly identical* copies of hardware cannot be made. By binding the application to the PUF, the software found on a platform will only function on that particular platform. One challenge of using PUFs is that many are noisy and have to be combined with fuzzy extractors (or similar) for their output to be usable [10, 17].

Of the suggested candidates, only a PUF would incorporate hardware intrinsic behaviour into the function F directly. When considering other choices of F there has been a need for some information to remain secret, e.g. encryption keys. If a PUF is chosen then the unclonable nature of PUFs ensures that neither F nor any other information must remain secret. The definition of a PUF states that simply knowing the design of the circuit and the input used is not enough to predict the output. The PUF would be incorporated into the design of the CPU and so by our assumptions it cannot be queried directly by the attacker. As the attacker is not able to directly query the PUF, the outputs from the PUF cannot be discovered by them. Using a PUF for the function F would meet with our requirements as its responses would be unpredictable and based on properties intrinsic to the device. Under our assumptions using a PUF as the function would securely bind the application to the device which it is loaded onto.

4.4 Implementation Considerations

We have proposed a technique to securely bind a piece of software to an individual hardware instance. However, there are several factors which would need to be considered when implementing this technique on a platform as well as the choice of function F.

Firstly, the masking and unmasking of operations/data would have to exist in the CPU only in order to prevent the scheme from being bypassed by an attacker. The unbinding must be a step in the execution process; as the operation/data is loaded from memory into the CPU it is unmasked before it is executed/used. If the masking is specific to a device and unmasking is a step in execution then programs copied from one hardware onto another would fail to execute correctly. An attacker will have to remove the binding from a piece of copied software before it could be executed correctly on a different piece of hardware.

Secondly, when choosing the function F it will be important to consider how to use the function securely and efficiently. For example, if F is chosen to be a block cipher is used with a 128 bit block size. If the device being protected uses 32 bit instructions then it will likely be more efficient to protect four instructions per challenge instead of one. Combining instructions may be required for the security of the scheme too. If 32 bit instructions were protected individually then each output of F is only one of 2^{32} possibilities. However the number of possible outputs of the function increases rapidly

[1]It is technically possible that two PUFs *could* have identical behaviour. However this is extremely unlikely, as a circuit would need to be created/found which always gave the same response as another circuit. Furthermore, even if by coincidence an attacker found a device 'x' that might have the same PUF behaviour as a device 'y', this would not help the attackers cause. The attacker wishes to make a large number of devices that have the same PUF behaviour as 'y' which will prove extremely difficult. We assume that our attacker cannot access a device with an identically behaving PUF as the platform it wishes to counterfeit.

if more instructions are protected simultaneously and so the output is one of 2^{64}, 2^{128} or more possibilities.

A further factor that would need to be considered when implementing the scheme is when and how the masking would be applied to the software. One approach would be for the software to be loaded to the device already masked. However this would require the manufacturer to interrogate the device in order to learn about the hardware intrinsic properties of that hardware instance. If this involved using an extra personalisation circuit that might introduce an attack surface for adversaries to use, it might also add a significant delay in the manufacturing process. Alternatively the device could add the masking when the software is loaded to it. This technique would prevent the manufacturer ever knowing the information about any particular hardware device, this may prevent insider attacks. However for the device to personalise itself it may require additional hardware increasing design complexity and the cost of producing each device.

4.5 Solution Analysis

A technique for preventing the counterfeiting of devices has been proposed in the preceding sections. This section will examine how it meets with the three requirements set out in Section 2.5.

1. **The proposed technique must grant the ability to securely personalise software to a hardware device.** The proposed technique provides the ability for a manufacturer to create a device-specific masking to bind software to hardware devices. The function F is used to provide device-specific mask values which are mixed with the instructions and data stored in the device memory. Several different candidate functions were proposed which would allow for different, secure personalisations for different devices. The suggested functions for F were described in Section 4.3 and suggestions for how they would be used in the proposed scheme were given. The security of the scheme was considered against two different attackers and an argument for the security of the scheme was included in Section 4.6.

2. **The software personalisation must make use of a hardware fingerprinting technique.** Of the three candidate functions proposed for F, only a PUF provides an adequate inclusion of hardware fingerprinting into the scheme by definition. However the alternative functions could also include the use of a PUF to provide a hardware fingerprint, which may be preferred if a PUF cannot be found which satisfies the unclonability requirement. A key programmed into the silicon of the CPU could be used to provide a device specific secret, however this would not provide the hardware intrinsicness required. Methods for including hardware intrinsic features when using block ciphers or hash functions were suggested when the different candidates for F were described in Section 4.3.

3. **The solution must protect all data that is to be stored in any of the storage areas that are present on the device.** The proposed technique would be implemented as an extension to the pipeline of instructions and data being loaded into the processor. Adding the technique as a pipeline stage would allow the instructions and data to be masked/demasked as they enter/exit the processor. In this setting there would be no data stored in memory in a non-protected manner.

4.6 Security Evaluation

This subsection will evaluate the security of the solution presented in this paper.

The proposed scheme uses an XOR operation to combine instructions with generated mask values. Due to the use of XOR, this scheme will remain secure only while two of the three elements involved (I_n, $F(C_n)$ and I'_n) remain secret. The values of I'_n are stored in the memory and will be available to the attacker. The values of I_n are the values sought by the attacker; these are only calculated within the CPU where the attacker cannot view them. To recover the values of the instructions I_n, the attacker must find or deduce the values of $F(C_n)$. The results of the function F on the values C_n are only calculated within the CPU, therefore, by assumption, the attacker is unable to read them. Therefore the security of the scheme is based on the strength of the function F.

The attacker described in Section 2.3 has the power to access all of the storage $MEM(x)$ of a platform x. Most of the challenges used with the function F are the masked instructions that are known to the attacker. The first challenge C_0 is also known to the attacker. It is therefore required that knowing only the input to F does not allow its output to be predicted. If the output of F is predictable then, using the challenges C_n, the attacker can recover the instructions I_n. If the attacker is unable to calculate the values $F(C_n)$, it is unable to either create a correct mask for its own application or unmask the instructions I'_n.

We now consider an attacker who is the developer of one of the multiple applications loaded onto a device x. This attacker has access to all of the masked instructions I'_n and their corresponding challenges C_n as well as a subset of the values for I_n which are the instructions comprising its own application. The developer-attacker uses its application and its masked version recovered from x to recover a set of corresponding C_n and $F(C_n)$ pairs. This poses a risk that the developer-attacker may be able to unmask a portion of any other applications found on the device x using the $C_n/F(C_n)$ pairs. Similarly the attacker might be able to use its knowledge to create the correct masking for any application of its choice. If the CPU were a 32 bit processor then the maximum application size would be 2^{32} addresses; however it is unlikely for an embedded application to be so large. If the applications were much smaller than 2^{32}, such as in the embedded setting, the chance of challenges for instructions of one application overlapping with instructions of another application significantly decreases. If this scheme were used in the multi-application setting, care would need to be taken by the manufacturer to choose a maximum application size that would keep the chance of significant challenge-space overlap between applications at an acceptable level. A countermeasure against challenge-space overlap would be to mask multiple instructions at the same time. If two operations were masked at once then the challenge space for a 32 bit processor would increase to 2^{64}, lowering the chance of challenges from applications overlapping one another. Overlapping of mask values used for binding software to hardware can also be

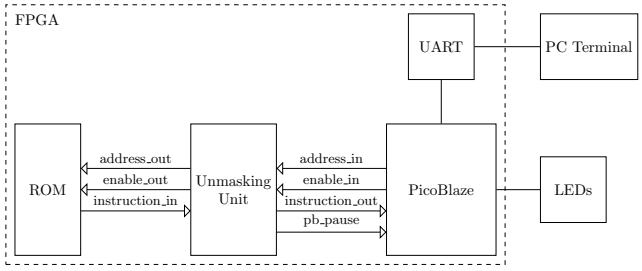

Figure 3: Diagram of implemented system.

decreased by ensuring that no two applications on a platform use the same C_0.

5. IMPLEMENTATION

A prototype implementation has been developed to demonstrate the viability of the scheme proposed in Section 4. The prototype also allows the performance impact of the scheme to be analysed.

The first half of this section describes the implementation including a system diagram of how the scheme was included into an example embedded system. Also described are the three different transitions that are possible between operations and how they are addressed by the prototype.

The second half of this section analyses the prototype implementation, describes the overheads of using the proposed scheme and also describes the costs associated with different choices for the function F.

5.1 Development Platform

The prototype implementation was designed and implemented on a Xilinx Spartan-6 FPGA SP601 evaluation board. VHDL was used to describe the circuits for the Unmasking Unit and other components in the prototype. The VHDL was synthesised for the Spartan 6 using the Xilinx ISE design suite. An Apple Macbook Pro was used for running the ISE design suite and synthesis tools as well as the terminal program PuTTY that read the serial output data transmitted by the prototype system.

The prototype implementation was developed for the FPGA board to create a real system operating using the proposed countermeasure. The implementation is similar to a normal embedded system, it contains a processor, a ROM that stores the application and IO devices. The IO devices used by the prototype are two LEDs and the serial output port of the evaluation board. The serial port functionality is used by the prototype implementation to output to a terminal program running on a laptop computer.

The processor included in the prototype system is the PicoBlaze^{TM} open soft-core processor developed by Xilinx and made freely available online [28]. The PicoBlaze^{TM} is attached to one of the block RAMs in the FPGA that is used as a ROM to store the program executed by the prototype. Example VHDL code for using the UART serial interface with a PicoBlaze^{TM} processor is provided with the processor and was used to connect the serial output to the system. A diagram of the prototype system is included in Figure 3, the Unmasking Unit is the element that was developed to implement the proposed scheme.

5.2 Demonstration Application

An example program was installed onto the prototype system that made use of the hardware devices attached to the PicoBlaze^{TM} system. This was so that the prototype would give an obvious indicator that it was executing correctly. The example program was written in PicoBlaze^{TM} Assembly Code and comprises two different stages of execution. Firstly the application runs through a set of instructions that use the UART serial output to transmit "Hello World!" to the laptop terminal application. This involved loading the characters to be outputted into a register and then outputting the value in the register to the UART data transfer component. Serial output from the prototype requires 2608 cycles to transmit one byte to the laptop, because of this it contains a buffer to store the data to be transmitted. However, the UART components buffer for data to be outputted over the serial line can store only a limited amount of data. Therefore, before sending data to be transmitted, the application checks the UART buffer to ensure that it is not full before adding the character to be outputted. This check is carried out by calling a simple procedure that reads from the UART transmit component and waits until the full flag is '0' (indicating that the buffer is not full). Once the UART buffer is not full the procedure finishes by executing a `RETURN` instruction.

The second half of the application ran on the prototype system increments a series of counters that are used to turn two LEDs on the evaluation board on and off. This part of the application comprises an infinite loop that repeatedly turns the two LEDs on and off. To ensure the LED flashing can be observed with the human eye a significant delay is required between each changing of the status of the LEDs. Three counters are used to create three loops with one loop executed inside another loop inside the third loop. Values of the two most significant bits of the third, outermost counter are assigned to the LEDs resulting in them following a off-off, off-on, on-off, on-on sequence.

5.3 Unmasking Unit

The scheme, as proposed in Section 4, was implemented using an Unmasking Unit positioned between the PicoBlaze^{TM} and the ROM components. For the prototype system the contents of the ROM were changed outside of execution and the masked values were loaded as the FPGA was programmed. The Unmasking Unit was developed to relay address, enable and instruction values between the PicoBlaze^{TM} and the ROM during execution and also to forward extra values in the case of the execution of `JUMP` or `CALL` instructions. The Unmasking Unit also calculates the mask ($F(C_x)$) values and on receipt of the masked $I'x$ instructions from the ROM it performs the required `XOR` operation before passing the resultant unmasked I_x instructions. The last function of the Unmasking Unit is to pause execution of the PicoBlaze^{TM} processor when required (this is described and explained in more detail later in this section). Due to the Unmasking Unit being entirely responsible for latching/fetching the masked instructions I'_x, the PicoBlaze^{TM} and FPGA RAM modules did not require any modification before being used in the prototype system (other than the data being stored in the RAM being changed). A diagram showing the inclusion of the Unmasking Unit is included in Figure 3.

The scheme implemented by the Unmasking Unit behaved as described in Section 4, a function of the contents of the previous memory address is used as a mask value for a

memory location. The function used in the prototype is a simple Linear Feedback Shift Register (LFSR). The LFSR is used to calculate the mask value $F(C_x)$ by setting the LFSR state to C_x and then generating three bits for the LFSR according to the taps used. The state of the LFSR after generating the three bits is used as the value $F(C_x)$ which is XOR'd with the masked instruction I'_x to reveal the instruction to be executed I_x.

The Unmasking Unit developed comprises three main processes as well as two much simpler pieces of logic that determined the values of the `enable_out` and `address_out` ports. During execution of the application being ran on the prototype system there are three situations that the Unmasking Unit may encounter. Firstly is the case of normal instruction execution: an instruction I_x is currently being executed and the next instruction to be executed is the following instruction I_{x+1}. In this case the Unmasking Unit will need to have stored the value I'_x such that I_{x+1} can be unmasked using $F(I'_x)$.

The second possible situation for the Unmasking Unit is that instruction I_x is a `JUMP` or `CALL` instruction. PicoBlaze^{TM} assembly code includes conditional and unconditional branch instructions, however they are both treated in the same way by the Unmasking Unit. In this situation the next instruction is either I_{x+1} (if the branch is not executed) or it is a different instruction I_y. To prepare for the possible branch instruction two features of the PicoBlaze^{TM} are exploited by the Unmasking Unit. The PicoBlaze^{TM} requires two cycles to execute an instruction, however the RAM can be accessed in only one cycle. To save power the PicoBlaze^{TM} only enables the RAM when it is in the second cycle of executing an operation, this is also when it sends the address of the next instruction to the RAM. Therefore there is a "spare" cycle available for the Unmasking Unit to use to load the extra data from memory it may require. In the example suggested previously this extra value would be I'_{y-1} and this would be read from the memory in the "spare" first cycle of the execution of I_x. To know the correct address for I'_{y-1} another feature of the PicoBlaze^{TM} is used. Most `JUMP` and `CALL` instructions include the address to be jumped to in the opcode; `0x36031` is the code for `JUMP NZ 031`, or if the `ZERO` flag is not set, then jump to address `031`. The address that may be branched to is therefore known to the Unmasking Unit so it is able to read the extra data it may need for unmasking the next instruction before the address of the next instruction required is known.

In the case of a `JUMP` or `CALL` instruction the enable and address values from the PicoBlaze^{TM} are not forwarded to the RAM. During the first half of the execution of an instruction the enable signal is 0, however the Unmasking Unit must transmit a 1 for the extra memory access. The address value from the PicoBlaze^{TM} will still be the address of the current instruction, however the Unmasking Unit will transmit an alternative value for the extra memory access.

A combinational process is used to handle changes on the instruction_in port of the Unmasking Unit. This process uses the calculated mask value to unmask the instruction read from the ROM. A sequential process, triggered by the system clock, is used to latch the instruction_in values and calculate the mask values used. The instruction latch process uses the enable signal to determine if the first or second half of the instruction is being executed, this is used when deciding if the value read from memory will create the next mask

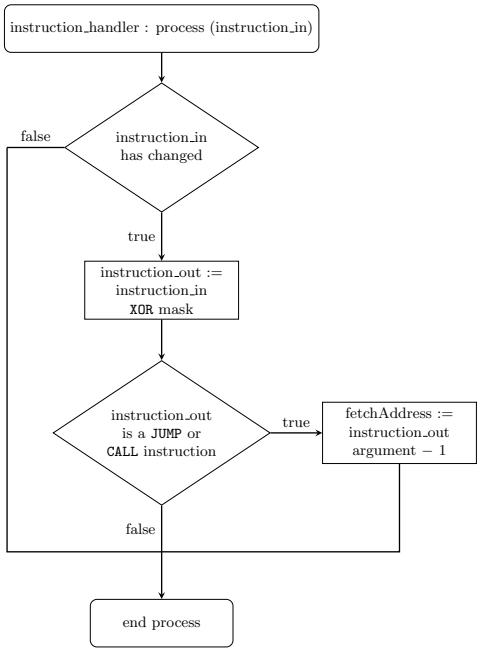

Figure 4: A flowchart of the instruction_in handling of the Unmasking Unit.

value or not. Diagrams of the operation of the combination instruction_in handler process and the clocked instruction latch process are found in Figures 4 and 5.

The third scenario the Unmasking Unit may encounter is when I_x is a `RETURN` type instruction or either a `JUMP@` or `CALL@` instruction. In this situation the address of the next instruction I_y is not known until halfway through the execution of I_x when the PicoBlaze^{TM} requests the next instruction. Before I_y is executed the Unmasking Unit will need to calculate the appropriate mask value $F(I'_{y-1})$, which will require a cycle before I'_y can be read and unmasked. Therefore the execution of the PicoBlaze^{TM} must be paused for one cycle to allow the Unmasking Unit to read the necessary data from memory and calculate the mask value required before loading, unmasking and outputting the next instruction. Therefore the Unmasking Unit includes an extra process that is responsible for detecting `RETURN` instructions and pausing the processor when required. A diagram of the operation of the pause process is included in Figure 6.

5.4 Performance Analysis

This subsection analyses the performance of the prototype presented above and also examines how other functions may be used as part of the scheme and the costs resulting from using a different function.

As described, the scheme has a performance impact on only one situation faced by the PicoBlaze^{TM}. Out of the 69 different instructions supported by the PicoBlaze^{TM} processor only 8 of the instructions require any delay at all before they can be executed. Furthermore, as it is only `RETURN` and the `JUMP@`/`CALL@` instructions that are slowed down by the scheme the impact on execution performance is likely to be minimal. The affected instructions are likely to be a small subset of the instructions which comprise a program. In a program there cannot be more `RETURN` instructions executed than

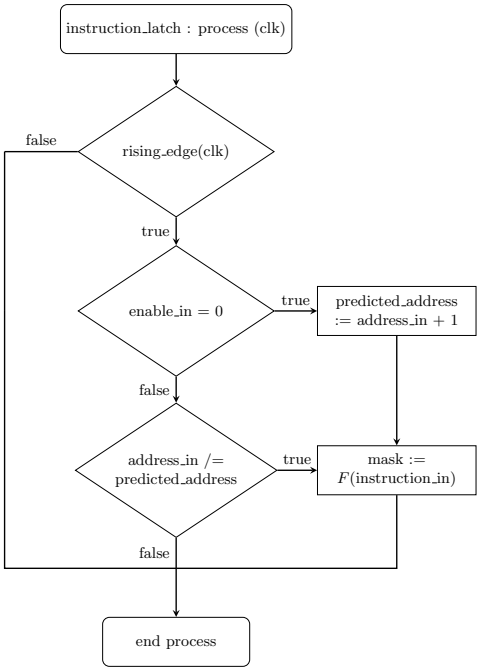

Figure 5: A flowchart of the calculation of the instruction mask values.

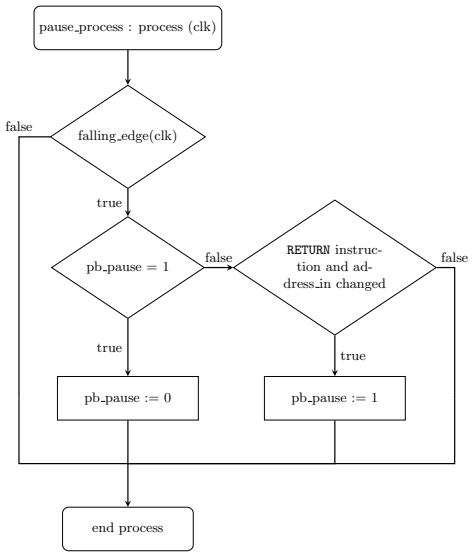

Figure 6: A flowchart of the operation of the PicoBlaze™ pause process.

`CALL` instructions. It is also reasonable to assume that data arithmetic, logic, input/output and loading/storing instructions make up a larger amount of any applications executed than the `RETURN` instructions. Considering the prototype system, the example program is comprised of 65 instructions, including only one `RETURN` instruction (although there are 14 `CALL` instructions). The procedure that is called is made up of 5 instructions including the `CALL` and `RETURN`. The impact of the scheme was to increase the execution time from 10 cycles to 11, an increase of 10%. However, the procedure that is called is part of the first half of the application that, including the block called, is made up of 52 instructions. Allowing for the procedure being called multiple times, the first half of the application requires 104 instructions to be executed, completing in 208 cycles. The first part of the application would execute in 222 cycles with the proposed scheme, therefore the actual performance penalty was only to increase execution time by 6.7%.

The prototype has been implemented using an LFSR as the function F. However the proposed scheme was designed to be able to use different types of functions for F, several candidates were described in Section 4.3. As described, the performance impact of using the scheme in the prototype was small. This was mainly due to exploiting features of the PicoBlaze™ in order to prevent processor execution being delayed. If the function required two cycles then the performance impact of the scheme would be greater as any `JUMP` instructions would also cause delays. Currently the scheme has 2 cycles for computation available in the first situation described in Section 5.3, 1 in the second and none in the case of `RETURN` instructions. Therefore if the function required 2 cycles to compute the mask value then there would still be no added delay if the next instruction is the next in memory. The delay from the PicoBlaze™ executing a `JUMP` would be 1 cycle, and a `RETURN` would require the processor to be paused for 2 cycles.

5.5 Security Analysis

The prototype was developed to demonstrate the viability of the proposed scheme, however, it does not completely meet the security requirements listed in Section 2.5.

Firstly, the prototype binds the software to the hardware by using an LFSR for the function F. This use of an LFSR meets neither the first or second requirements described in Section 2.5. The first requirement demands that the personalisation be secure, however the attacker is able to emulate the operation of the LFSR and calculate the mask values used because the prototype does not include any information the attacker would not know.

Requirement 2 requires that the software personalisation make use of a hardware fingerprinting technique. This has not yet been implemented on the prototype and is a topic for future work on the prototype implementation. Hardware Intrinsic Security will likely be included by using a PUF as described in Section 3.2, however an alternative technique may also be suitable. Section 4.3 described how many different functions might be appropriate for use with the proposed binding technique.

The third requirement was that the solution must protect all of the device data. This has been met because the only data stored on the prototype is the application data that has been bound to the implementation. The current version of the Unmasking Unit does not have the ability to mask/unmask

data which is stored/loaded in the device memory. Therefore, if any data were saved to or read from the memory it would currently be unprotected. However, the prototype has demonstrated that the proposed technique may be used for protecting instructions stored in memory and so could likely also be applied to storing data in memory. The current Unmasking Unit is able to access areas of memory without disrupting execution of the process (as is used for unmasking instructions after a jump). It may be possible that the extra loading from memory can be completed without significantly affecting the execution of the PicoBlaze^{TM}. However this is a problem to be addressed in future work.

6. CONCLUSION

In this paper we have presented a novel technique that binds applications to the platform they are installed onto. The proposed technique can be used to prevent unauthorised modification of device firmware by ensuring that applications cannot be copied from legitimate platforms and then executed on devices other than the device the firmware was provisioned for. Hardware intrinsic properties of the device could be used in the binding which personalise the software to only the particular hardware instance on which it is installed. A proof-of-concept implementation of the scheme has been presented which includes the proposed scheme and demonstrates that the scheme can be deployed with only a small performance overhead.

The solution proposed securely binds software to a hardware device and protects the data stored on the platform wherever it is stored. The binding is secure against even a powerful attacker who can read any device storage outside of the CPU at any time.

Future work on the prototype would be to implement the scheme for protecting data which is stored in the memory, not just the application instructions. Another extension for the prototype would be to include a hardware fingerprinting technique as part of the generating of the instruction masks, this would ensure that the masking is specific to the hardware device. Further work on the scheme would be to consider how regularly the initial C_0 mask values would need to be updated in order for the scheme to be secure. Another area for study would be to consider how firmware updating could be performed if this scheme were to be used. Provisioning a different version of the firmware for every single legitimate device might not be achievable and so a specific update protocol may be needed.

7. ACKNOWLEDGEMENTS

Robert P. Lee is supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/K035584/1).

8. REFERENCES

[1] AGMA. AGMA Global - Elimination of Counterfeiting, 2015.

[2] B. Anckaert, M. Madou, B. D. Sutter, B. D. Bus, K. D. Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. pages 15–20, 2007.

[3] A. Appel. Deobfuscation is in NP. *Princeton University, Aug*, 21:2, 2002.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (Im)possibility of Obfuscating Programs. In J. Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2001.

[5] C. S. Collberg and C. D. Thomborson. Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection. *IEEE Trans. Software Eng.*, 28(8):735–746, 2002.

[6] P. F. Cortese, F. Gemmiti, B. Palazzi, M. Pizzonia, and M. Rimondini. Efficient and Practical Authentication of PUF-based RFID tags in Supply Chains. In *RFID-Technology and Applications (RFID-TA), 2010 IEEE International Conference on*, pages 182–188, June 2010.

[7] K. B. Frikken, M. Blanton, and M. J. Atallah. Robust Authentication Using Physically Unclonable Functions. In P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, editors, *Information Security, 12th International Conference, ISC 2009, Pisa, Italy, September 7-9, 2009. Proceedings*, volume 5735 of *Lecture Notes in Computer Science*, pages 262–277. Springer, 2009.

[8] B. Gassend, D. E. Clarke, M. van Dijk, and S. Devadas. Silicon Physical Random Functions. In V. Atluri, editor, *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS 2002, Washington, DC, USA, November 18-22, 2002*, pages 148–160. ACM, 2002.

[9] M. A. Gora, A. Maiti, and P. Schaumont. A Flexible Design Flow for Software IP Binding in Commodity FPGA. In *IEEE Fourth International Symposium on Industrial Embedded Systems, SIES 2009, Ecole Polytechnique Federale de Lausanne, Switzerland, July 8-10, 2009*, pages 211–218. IEEE, 2009.

[10] J. Guajardo, S. S. Kumar, G. J. Schrijen, and P. Tuyls. FPGA Intrinsic PUFs and Their Use for IP Protection. 4727:63–80, 2007.

[11] J. Guajardo, S. S. Kumar, G. J. Schrijen, and P. Tuyls. Brand and IP protection with Physical Unclonable Functions. In *International Symposium on Circuits and Systems (ISCAS 2008), 18-21 May 2008, Sheraton Seattle Hotel, Seattle, Washington, USA*, pages 3186–3189. IEEE, 2008.

[12] S. Katzenbeisser, Ü. Koçabas, V. Rozic, A. Sadeghi, I. Verbauwhede, and C. Wachsmann. PUFs: Myth, Fact or Busted? A Security Evaluation of Physically Unclonable Functions (PUFs) Cast in Silicon. 7428:283–301, 2012.

[13] T. Kean. Cryptographic Rights Management of FPGA Intellectual Property Cores. In *FPGA*, pages 113–118, 2002.

[14] S. Kleber, F. Unterstein, M. Matousek, F. Kargl, F. Slomka, and M. Hiller. Secure Execution Architecture based on PUF-driven Instruction Level Code Encryption. *IACR Cryptology ePrint Archive*, 2015:651, 2015.

[15] P. Koopman. Embedded System Security. *IEEE Computer*, 37(7):95–97, 2004.

[16] J. W. Lee, D. Lim, B. Gassend, G. E. Suh, M. van Dijk, and S. Devadas. A Technique to Build a Secret Key in Integrated Circuits for Identification and Authentication Applications. In *VLSI Circuits*, pages 176–179. Widerkehr and Associates, 2004.

[17] R. Maes. *Physically Unclonable Functions - Constructions, Properties and Applications*. Springer, 2013.

[18] R. Maes, P. Tuyls, and I. Verbauwhede. Intrinsic PUFs From Flip-Flops on Reconfigurable Devices. *Benelux Workshop on Information and System*, (71369):1–17, 2008.

[19] P. Marwedel. *Embedded System Design*. Kluwer, 2003.

[20] T. Morris. Trusted Platform Module. In H. C. A. van Tilborg and S. Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 1332–1335. Springer, 2011.

[21] I. Paul. New tool reawakens disabled hardware in high-end AMD Radeon graphics cards, 2015.

[22] Postscapes Labs. Internet of Things Market Size, 2015.

[23] E. Simpson and P. Schaumont. Offline Hardware/Software Authentication for Reconfigurable Platforms. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 2006.

[24] G. E. Suh and S. Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. pages 9–14, 2007.

[25] T. Toyofuku, T. Tabata, and K. Sakurai. Program Obfuscation Scheme Using Random Numbers to Complicate Control Flow. In T. Enokido, L. Yan, B. Xiao, D. Kim, Y. Dai, and L. T. Yang, editors, *Embedded and Ubiquitous Computing - EUC 2005 Workshops, EUC 2005 Workshops: UISW, NCUS, SecUbiq, USN, and TAUES, Nagasaki, Japan, December 6-9, 2005, Proceedings*, volume 3823 of *Lecture Notes in Computer Science*, pages 916–925. Springer, 2005.

[26] Trusted Computing Group. Part 1 Design Principles. In *TPM Main Specification*. TCG, 2011.

[27] V. van der Leest and P. Tuyls. Anti-Counterfeiting with Hardware Intrinsic Security. pages 1137–1142, 2013.

[28] Xilinx, Inc. PicoBlaze 8-bit Microcontroller, 2011.