Special Issue on Cryptographic Engineering for IoT: Ensuring Secure Application Execution and Platform Specific Execution in Embedded Devices

ROBERT P. LEE, ISG Smart Card and IoT Security Centre, Royal Holloway, UK
KONSTANTINOS MARKANTONAKIS, ISG Smart Card and IoT Security Centre, Royal Holloway, UK
RAJA NAEEM AKRAM, ISG Smart Card and IoT Security Centre, Royal Holloway, UK

The Internet of Things (IoT) is expanding at a large rate, with devices found in commercial and domestic settings from industrial sensors to home appliances. However, as the IoT market grows, so does the number of attacks made against it with some reports claiming an increase of 600% in 2017. This work seeks to prevent code replacement, injection and exploitation attacks by ensuring correct and platform specific application execution. This combines two previously studied problems: secure application execution and binding hardware and software. We present descriptions of both problems and requirements for ensuring both simultaneously. We then propose a scheme extending previous work that meets these requirements, and describe our implementation of the soft-core Secure Execution Processor developed and tested on Xilinx Spartan-6 FPGA. Finally, we analyse the scheme and our implementation according to performance and the requirements listed.

CCS Concepts: • **Security and privacy → Embedded systems security**; **Hardware security implementation**; *Tamper-proof and tamper-resistant designs*; Digital rights management;

Additional Key Words and Phrases: Internet of Things, Secure Application Execution, Platform Specific Execution, Hardware-Software Binding

ACM Reference Format:
Robert P. Lee, Konstantinos Markantonakis, and Raja Naeem Akram. 2019. Special Issue on Cryptographic Engineering for IoT: Ensuring Secure Application Execution and Platform Specific Execution in Embedded Devices. 1, 1 (May 2019), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the advent of the Internet of Things (IoT), embedded systems are being included in an increasing number of devices. The IoT market is growing at a huge rate with some forecasts estimating it will be worth as much as $457B in 2020 [6]. Symantec found a 600% increase in attacks against IoT devices in 2017 [24]. Therefore, more efforts to secure IoT devices are needed.

Many of the attacks on IoT devices and other systems attempt to gain control of a device by exploiting bugs to run arbitrary program code. A counter to these attacks is to include measures to ensure Secure Application Execution (SAE). SAE secures application control flow to prevent arbitrary code execution [12].

Authors' addresses: Robert P. Lee, ISG Smart Card and IoT Security Centre, Royal Holloway, Egham Hill, Egham, Surrey, TW20 0EX, UK, robert.lee.2013@live.rhul.ac.uk; Konstantinos Markantonakis, ISG Smart Card and IoT Security Centre, Royal Holloway, Egham Hill, Egham, Surrey, TW20 0EX, UK, k.markantonakis@rhul.ac.uk; Raja Naeem Akram, ISG Smart Card and IoT Security Centre, Royal Holloway, Egham Hill, Egham, Surrey, TW20 0EX, UK, r.n.akram@rhul.ac.uk.

XXXX-XXXX/2019/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Related to SAE is the problem of Platform Specific Execution (PSE) or securely binding hardware and software together. This problem seeks to ensure only applications from authorised parties can execute. However, the two problems differ in two main ways. Firstly, SAE ensures the current application is executing correctly, but not the legitimacy of the author of the code [9]. Conversely, PSE focusses on ensuring only applications personalised to a device by an authorised party can execute but does not consider legitimate applications failing to operate correctly [16]. Therefore, securing IoT devices requires a combination of both problems: ensuring devices correctly execute legitimate applications and nothing else.

1.1 Contributions

The main contributions of this paper are:

(1) Modelling a new problem combining hardware/software binding with Secure Application Execution (Section 2).
(2) Providing a transition based perspective of Secure Application Execution (Section 2.2).
(3) Proposing a set of requirements for securing application control flow and execution location (Section 2.4).
(4) Proposing a scheme to solve the problem considered (Section 4).
(5) Implementing the proposed scheme and describing the components of the developed system (Section 5).
(6) Analysing the scheme and implementation (Section 6).

1.2 Paper Structure

The paper is structured as follows: Section 2 contains an overview of the problem considered in this paper. Section 3 describes the related work in hardware/software binding and Secure Application Execution. Section 4 contains the solution proposed for securing application confidentiality and execution. Section 5 describes the implementation of the proposed scheme and includes a breakdown of the main engineering decisions and constraints. Section 6 analyses the proposed scheme and implementation. Section 7 concludes the paper and describes potential future work in this area.

2 PROBLEM DESCRIPTION

This paper considers ensuring secure, unmodified and platform specific execution of applications. Informally, if an application, *App*, has finished executing, we guarantee it has executed correctly and on the specific hardware device it is intended for. In this work, correct execution is defined as following a legitimate control flow with no operations modified or omitted from the program as it was installed. The following subsections define Platform Specific Execution (Section 2.1), Secure Application Execution (Section 2.2), the attacker model considered (Section 2.3) and the requirements for a proposed solution (Section 2.4).

2.1 Platform Specific Execution

Platform Specific Execution (PSE) is the problem of ensuring a Software instance SW installed on a Device (D) is able to execute on D and on no other device, D'. Usually, an SW is deployed to multiple devices or systems. Therefore to achieve PSE, each SW must be made unique for the D it is executed by.

PSE can be guaranteed by using hardware-software binding to securely personalise a software instance, SW, to many devices. With this technique an installed software, SW_D is made reliant on some element or information held or known only by D, the result of this being that SW loaded onto a different device, D', will not execute correctly. Therefore a hardware device is restricted to

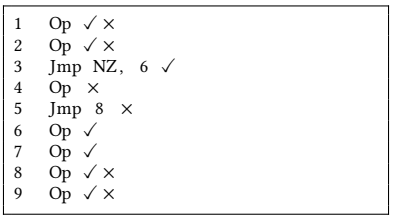

Fig. 1. Legitimate executions follow each instruction in a control path such as the ticked instructions, illegitimate executions skip instructions such as the path avoiding the `JUMP NZ, 6` that is marked by crosses.

executing only the *SW* bound to it and no different *SW′*. This can prevent code injection attacks. Hardware-Software two-way bounds have been used previously to prevent counterfeiting, firmware modification and for securing software in smart cities [16, 17].

2.2 Secure Application Execution

As well as PSE, we are concerned with Secure Application Execution (SAE), which is defined as ensuring applications only follow correct control flows as defined by the developer [1]. A correct control flow has been followed if every instruction has been executed without any being skipped or modified. While instructions not executed due to correctly executed conditional `JUMP` instructions have been legitimately skipped, instructions not executed due to modification or deletion are illegitimately skipped and compromise SAE. Therefore, achieving SAE comprises two main requirements: ensuring instruction integrity and ensuring Control Flow Integrity (CFI). PSE considers instruction integrity, therefore this subsection will focus on CFI.

Examples of legitimate and illegitimate execution of an application are found in Figure 1. The figure shows a code snippet with instructions marked with ticks (✓) and crosses (×). A legitimate execution through the snippet would be to execute every instruction marked with a tick, when instruction 3 was executed, the zero flag was not set and control flow jumps to instruction 6. If only the instructions marked with crosses were executed, this would be an illegitimate execution because the conditional `JUMP` of instruction 3 is entirely avoided.

In this work, the legitimacy of logical tests based on external data is not included in the scope of SAE. For example, attacks tampering with sensors/sensor data to ensure one side of an `if` instruction is always chosen is out of scope. However, skipping a conditional `JUMP` operation to force execution of one path of an `if` statement, as shown in Figure 1, is within the scope of this work. We also include attacks attempting to force control flow to alternative portions of the application (perhaps ahead of schedule or to code not in the current flow).

The example of illegitimate execution shown in Figure 1 shows one type of instruction transition being attacked. To ensure SAE, we argue there are three types of instruction transitions to protect. These are: sequential transitions, instruction-dependent transitions and instruction-independent transitions. Each of these are described below with examples of attacks.

Sequential transitions are the simplest and most common instruction transition. They occur when an instruction at location x is executed and the following instruction to be executed is at location $x + 1$. The program counter increases by 1 and no jump occurs. These are found after arithmetic, logical, I/O or load/store instructions. However, they also occur after conditional `JUMP` instructions whenever the logic test fails and the jump is not executed. The example of illegitimate execution in Figure 1 shows the sequential transition between instructions 2 and 3 changed to prevent instruction 3 executing.

Instruction-dependent transitions happen after unconditional `JUMP` or `CALL` instructions, if the destination of the `JUMP` is determined by the operand of the instruction. For example, `JUMP x` or `CALL x` causes an instruction-dependent transition by changing the value of the program counter (the location of the next instruction) to `x`. The transition from instruction 5 to instruction 8 in Figure 1 is an example of an instruction-dependent transition. Relative `JUMP` instructions are also considered as causing instruction-dependent transitions, as the distance jumped and instruction location can be used to replace the instruction with an equivalent, absolute `JUMP` instruction.

Finally, instruction-independent transitions occur when the next instruction is not identifiable from the instruction only. The most common causes of instruction-independent transitions are `RETURN` and conditional `JUMP` instructions. A `RETURN` can transition to different locations based on where the function was called from, the location of the next instruction is loaded from the stack and not from the current instruction. This allows functions to be called from different locations and execution flow to return to where the function was called from after execution. However, it also introduces a weakness exploited by attacks such as Return Oriented Programming (ROP) [4]. Conditional `JUMP` instructions lead to the immediately proceeding instruction or a remote instruction depending on the processor state at execution time. Therefore, the instruction alone is insufficient to identify the location of the next instruction to execute as processor flags, such as the Zero or Carry flags, are needed. In the legitimate execution in Figure 1, the transition from instruction 3 to instruction 6 is an instruction-independent transition. The `JUMP` was chosen because the Zero flag was not set.

If all three types of instruction transitions are secured, it follows that SAE has been achieved as it is not possible for program flow to deviate from the designed path.

2.3 Attacker Model

The attacker considered in this work is a Dolev-Yao attacker [10] able to read and manipulate data but unable to break cryptographic primitives. They can read and write to any memory location exterior to the CPU, such as RAM or long-term storage, but they are unable to modify CPU cache memory or registers directly. The attacker has access to personalised applications extracted from legitimate devices but does not have access to the original application binary. This simulates the attacker buying the device protected with the SAE-PSE scheme and extracting the memory contents.

We assume the device is assumed to have protection mechanisms preventing side-channel leakage and so the attacker is not able to use side-channel vulnerabilities such as power analysis [14], acoustic side-channels [11] or cache timing side-channels [13, 18] to attack the proposed scheme.

The attackers goal is to achieve any of the following:

(1) Skip the execution of an instruction without detection (execution must continue as if no instruction has been skipped).
(2) Modify an instruction on the device that successfully executes.
(3) Redirect software control flow to an alternative portion of code from memory. The code must execute earlier than designed or else appear in the control flow when it would have not appeared at all.
(4) Execute software taken from one device, D, on a device, D' where $D \neq D'$.
(5) Execute unauthorised software written by the attacker on a device D.
(6) Unmask software taken from a device D, in order to discover the original, unmasked, source binary.

2.4 Requirements

The requirements for a solution to the problem considered are:

- **R1) Software Confidentiality:** the software must not be revealed to attackers. When stored in memory, the software must be encrypted.
- **R2) Authorised Execution:** only applications bound to a device can be executed by the device.
- **R3) Immutable Applications:** securely personalised software must not be meaningfully modifiable by unauthorised parties. Attackers are unable to make targeted changes to the software.
- **R4) SAE - Secure Sequential Transitions:** sequential transitions where no `JUMP`, `CALL` or `RETURN` occurs must be secured in the proposed scheme.
- **R5) SAE - Secure Instruction-dependent Transitions:** instruction-dependent transitions must be protected to ensure that after the transition, the program counter value can only be the value determined by the instruction.
- **R6) SAE - Secure Instruction-independent Transitions:** instruction-independent transitions must result in the program counter being changed to the appropriate value after a conditional `JUMP` or the address following the last executed `CALL` if the transition is caused by a `RETURN`.

3 RELATED WORK

This section considers previous work in Platform Specific Execution and Secure Application Execution and analyses it against the requirements listed in Section 2.4.

3.1 Platform Specific Execution

Platform Specific Execution is also known as binding hardware and software together and has been applied to several problems including: counterfeiting prevention, securing device firmware, protection of smart cities and in Digital Rights Management (DRM) [15–17]. In 2003, Krasinski and Rosner patented a method for binding a piece of software to a specific hardware instance. Their method created a unidirectional bound to attach a piece of DRM software to a device to protect digital audio or visual content. A key is derived from unique or distinctive hardware, software or firmware identifiers to ensure only the specific hardware instance may access the protected media [15].

Later, in 2008, Atallah et al. published a mechanism for binding software to a specific hardware instance in a Virtual Machine environment. Like Krasinski and Rosner, Atallah et al. created a unidirectional bond to attach software to hardware. To prevent distinctive hardware, software or firmware features being virtualised, Atallah et al. used a Physically Unclonable Function (PUF) to confirm the presence of legitimate hardware to the executing software [2].

Lee et al. were the first authors to propose the need for a bi-directional bound between hardware and software in 2016. They argued for the need to connect both to ensure software executes only on legitimate hardware and also that hardware only executes legitimate software, which would ensure the integrity of deployed products as neither the software nor the hardware could be inauthentic. The authors also suggested the use of a PUF to provide a device-specific secret for use in creating the bound [16]. Lee et al. produced their bound between hardware and software by masking each instruction using the previous stored instruction as well as a device-specific secret which ensures the application will execute correctly on no other device and that applications bound to the device are able to execute correctly. Using the previous instruction in memory prevents attackers creating alternative, executable applications by moving instructions. It also prevents the masking from behaving as an electronic code book, a block cipher mode with known flaws [16]. A diagram of the Lee et al. unmasking process is included in Figure 2.

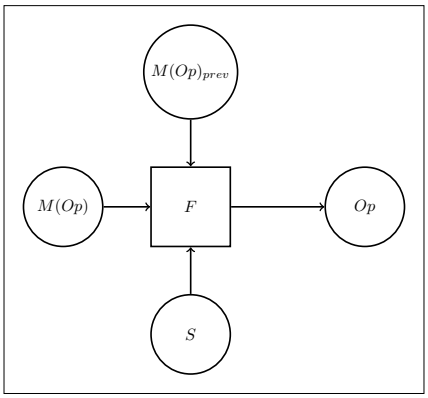

Fig. 2. Lee et al. binding uses a secret, S, and the previous, masked instruction, $M(Op)_{prev}$, to unmask $M(Op)$ and reveal Op for execution.

Requirements R2 and R3 make it necessary for the PSE scheme used in the proposed solution to be bidirectional. The Krasinski-Rosner and Atallah et al. schemes are therefore not suitable for inclusion. The Lee et al. scheme provides a two way bound sufficient for the two requirements, however like the Krasinski-Rosner and Atallah et al. schemes, it offers no guarantees about application execution. These would need to be included by extension or modification of the original proposal.

3.2 Secure Application Execution

A significant work in SAE is the 2005 paper by Abadi et al. introducing CFI enforcement as a means of preventing various attacks. Prior to their work, most schemes were developed to protect against individual attacks, however Abadi et al. proposed a method for protecting existing code without hardware changes. Their technique uses a Control Flow Graph (CFG) drawn from an application binary using static analysis. Security of execution is ensured by only allowing control flow transfers included in the CFG [1].

In 2017, Clercq and Verbauwhede surveyed CFI literature and considered the problem using CFGs and what they labelled as backward and forward edges [9]. This is similar to our identification of instruction transitions, however Clercq and Verbauwhede focus on instructions causing control flow transfers and not on sequential transitions. Repeated sequential transitions are merged into CFG nodes and it is assumed the single entry/exit nature of basic blocks holds. The authors describe *forward edges* as control flow transfers caused by jumps and calls and *backward edges* as those caused by returns [9]. One disadvantage of considering all jumps and calls equally is it loses the distinction between jumps to register-determined and instruction-determined locations. This is significant as, when implementing a scheme, instruction-dependent transitions are simpler to process than instruction-independent transitions, as described by Lee et al. [16]. Therefore, considering transitions is important when also considering hardware-software binding and creating schemes for implementation.

The rest of this section describes some proposed methods relevant to this work for ensuring CFI.

3.2.1 Jump Labelling. Jump labelling is a CFI protection technique proposed by Abadi et al. in 2005 [1]. They suggested it as a defence, implementable in software, for protecting against various attacks. The mechanism works by labelling the destinations of control flow instructions with IDs checked as part of the jump so that if the label at the destination fails to match that at the jump then program execution is aborted [1].

It was originally proposed for execution in software only [1], however it can be implemented in hardware and was by Christoulakis et al. and Sullivan et al. in later works [5, 23]. However, the labelling approach has several issues preventing it from meeting Requirements R4 to R6. Firstly, it does not protect instructions executed after the jump instructions, it only secures the jump instructions. Secondly, jump labelling relies on the attacker being unable to interfere with the program in memory, while our attacker can write to memory (Section 2.3). Finally, it is vulnerable to attacks changing a return address from the original address to an alternative address. These attacks are described in more detail in Section 4.2.

HAFIX extends on the labelling approach by maintaining a table of labels and marking them active when a call is made from a function. On the return instruction, the table is consulted to ensure the label is active. An error occurs whenever an execution attempts to return to an inactive function. This approach lends itself to hardware implementation and could be included in the processor pipeline. Overall, the cost of HAFIX is small as it merely requires a one bit flag for every call instruction in the program [7]. However, HAFIX does not protect all transitions and does not protect the instruction jumped to, so it does not meet Requirements R5 and R6.

3.2.2 Shadow Call Stacks. A Shadow Call Stack (SCS) is a mechanism used to protect return operations from being misused by attackers. It is effective against ROP or return-into-libc attacks and is regularly included with jump/call protections in CFI schemes. Essentially, an SCS is a second stack storing the return addresses added to the "normal" stack. The SCS is checked when a return instruction executes to ensure the top of both the SCS and the "normal" stack are equal. SCS is simple to implement as it only requires extending the logic of `RETURN` operations and does not require performing program analysis [9].

As identified by Clercq et al. in 2017, the location of the SCS is critical to the security of the scheme. A stack in main memory would allow a large call depth but would be vulnerable to attackers controlling memory. Different approaches suggest using dedicated hardware buffers or memory only accessible by `CALL` and `RETURN` instructions or using memory divided on a kernel level to store the SCS [9]. Therefore, SCS meets Requirement R6 but not Requirements R4 and R5.

3.2.3 SOFIA. A significant, recent work in the area of SAE is SOFIA, a software and control flow integrity architecture proposed by Clercq et al. in 2017 [8]. They propose a method for securing CFI and describe the performance of an implementation developed. This work sets five goals for protecting application execution: software integrity, control flow integrity, tampered code protection, code confidentiality and reverse engineering protection. These are chosen to ensure applications correctly execute and to minimise attackers ability to examine the code to look for flaws/vulnerabilities [8].

In essence, SOFIA protects application execution by ensuring attackers are unable to change any instructions. Furthermore, the architecture creates location-specific protections for each of the instructions to ensure they cannot be moved to different locations in memory and be executed in a different order. Integrity checks are included to detect modified instructions before execution [8].

The instruction protection mechanism in SOFIA is partly in the form of a masking scheme such as that proposed by Lee et al. in 2016 [16]. However, instead of chaining instructions, Clercq et al. have used a nonce and the preceding and current Program Counter (PC) values as the information used by the function to protect instructions [8]. As a result, the instruction masking calculation is as follows (adapted from [8]):

$$Op_x = Op'_x \oplus E_{k_1}(\omega||x-1||x)$$

Where x is the PC value of the current instruction, $x-1$ is the PC value of the previously executed instruction, ω is a nonce and Op_x and Op'_x are the instruction and masked instruction in location x respectively. Also calculated with the decrypted instructions are blockwise CBC-MAC values used as an integrity check using a second key k_2. The MAC of each block is encrypted and then stored in the first location of each block and is loaded and compared with the calculated block MAC as the instructions are decrypted. The combination of these checks provides an integrity check ensuring attacks on the stored instructions are detected and control flow specific instruction masking prevents instructions being moved in the application [8].

One strength of SOFIA is it considers a powerful attacker with full control of external storage, I/O and buses [8]. However, one disadvantage of SOFIA is how it handles reused code accessed from multiple locations. Due to the instruction encryption, each instruction can only be accessed from one location. To allow code to be accessed from multiple locations, Clercq et al. mark some blocks as accessible from multiple locations. These special blocks start with a series of entry points, one for each `JUMP` or `CALL` leading to the block. Each of these entry points are modified instructions allowing the calculated MAC and instruction decryption values to each be computed from a designated "first" instruction in the block instead of the actual first instruction in the block [8]. This is rather inflexible and requires code analysis to be performed to identify all the locations each function is called from before appropriate entry points can be appended. Therefore, while SOFIA meets Requirements R1 to R6 it is not a perfect solution as handling reused code requires extra program analysis and blocks of code adding to each function based on the number of locations calling the function.

4 PROPOSED SOLUTION

This section describes the proposed solution to the problem posed in Section 2. The solution proposed is based on the hardware-software binding scheme proposed by Lee et al. in 2016 [16] described in Section 3.1. However, that only meets some of the requirements from Section 2.4. Lee et al. listed and met requirements similar to Requirements R1 to R3 when proposing their binding scheme. However, extensions to provide SAE and meet Requirements R4 to R6 are needed. The scheme proposed in this section will extend the Lee et al. scheme using ideas from schemes listed in Section 3.2.

4.1 Protecting Sequential Transitions

The first type of instruction transition defined in Section 2.2 is sequential transitions. Transitions of this type are incidentally and implicitly protected by the Lee et al. scheme due to the cascading nature of the hardware-software bound. An instruction Op stored in location x is masked using the data stored in location $x-1$ (the previous, masked instruction) to produce Op'. For efficiency, the authors reduce the memory operations needed for unmasking by latching data read from memory after the instruction is unmasked. This allows, in the case of sequential transitions, masked data to be used in unmasking the next instruction without being loaded again. This latching allows the Lee et al. scheme to provide implicit sequential transition security.

Extra memory accesses are only required in the Lee et al. scheme in the case of control flow instructions. Therefore, the only helper data available in unmasking an instruction after a sequential transition is the previously executed instruction. This provides implicit sequential transition security and consequently, instructions in series execute correctly and can only be modified by someone who knows the masking key. Therefore, the Lee et al. scheme protects instructions executed in series by guaranteeing sequential transitions.

```
    Op
    Op
    Jmp x
                .
                .
                .
    Op
x:  Input r1, SecurityToken
    Check r1
    Load r2, SensitiveData
    Output r2
    Op
                .
                .
                .
```

```
    Op
    Op
    Jmp x
                .
                .
                .
    Check r1
x:  Load r2, SensitiveData
    Output r2
    Op
    Op
    Op
                .
                .
                .
```

Fig. 3. When protected by only the Lee et al. scheme, the code snippet on the left can be modified to leak sensitive data by moving code into a location reached by a `JUMP` operation.

4.2 Protecting Instruction-Dependent Transitions

While the Lee et al. scheme protects sequential transitions between instructions, it does not secure instruction-dependent transitions. These can be manipulated by an attacker to execute arbitrary program code and violate Requirement R3, an example is included in Figure 3. Consider the application on the left side of Figure 3, execution starts from the top and executes two arbitrary instructions followed by a `JUMP`. The `JUMP` leads to a short block of code, x, that performs a security check before outputting sensitive data. The Lee et al. scheme only secures the `JUMP` instruction that sets the PC value to x, but not the *contents* of x. Therefore, it is possible for the code to be shifted in memory to skip the security check before outputting the sensitive data. The attacker could change the code into that on the right of Figure 3 to output the sensitive data without executing the security check. Eventually, the attack may be discovered when execution reached $x - 1$ and the instruction unmasked incorrectly, however this might happen long after the leak, or never.

We therefore propose modifying the scheme of Lee et al. to prevent instructions from being moved from the locations they were originally installed in. The proposed modification is to combine the Lee et al. scheme with SOFIA by Clercq et al. (described in Section 3.2.3). In SOFIA, an instruction is masked using the PC value of the instruction, the PC value of the previous instruction and secret information. This produces a device *and* memory location-specific masking preventing instructions from being moved. If an instruction at location y were moved to location x it would no longer unmask correctly because the unmasking would use x and $x - 1$ instead of y and $y - 1$. Using the location in calculating the instruction masking value prevents instructions from being moved. SOFIA introduces the need for a tree-like structure at function entry adding a new instruction for each call to the function. However, as the jump source is protected by the application binding (Section 4.1), preventing instruction movement is the only additional property needed. The new masking equation will be as follows:

$$Op'_x = F(Op_x, Op'_{x-1}, x)$$

Where x is a memory location or PC value, Op_x is the instruction in location x in the application, Op'_x is the masked version of Op_x and F is the function used to unmask Op'_x. This extension binds each instruction to the specific device and also the specific memory location it is to be executed from. This ensures attacks such as demonstrated in Figure 3 are no longer possible.

4.3 Securing Instruction-Independent Transitions

After combining elements from the Lee et al. and Clercq et al. schemes, the result is a masking scheme ensuring code immutability and that control flow instructions lead to the *instructions*

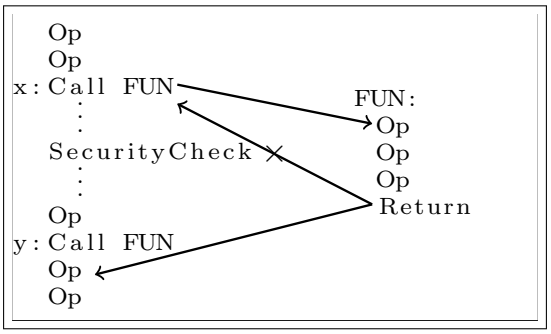

Fig. 4. Changing return addresses can be used to skip security critical instructions.

intended. However, there is still a weakness associated with function code accessed by CALL and RETURN-type operations. The weakness is that while the scheme ensures that CALL instructions lead to the correct code it does not ensure CALL and RETURN instructions are executed in correct pairs.

Consider the code example in Figure 4. A function is called from two locations, x and y, and a security check is computed between the calls. As stated in Section 2.3, the attacker can interfere with any memory location including the stack. Therefore, the check can be avoided by waiting until FUNC is called and then changing the return address on the stack from $x + 1$ to $y + 1$. This avoids executing the security check and allows unauthorised access to later parts of the application. This attack is possible in the combined Lee et al.-Clercq et al. scheme as no instructions are modified.

Therefore, a mechanism is required to ensure the destination of RETURN instructions matches (+1) the location of the CALL transferring control flow to the function. We propose an instruction set extension to transform CALL and RETURN into two part instructions. This change adds CALL−IN and RETURN−OUT to the instruction set. These instructions must be executed before CALL and after RETURN instructions respectively to prevent attacks as shown in Figure 4. The check will be to store a value provided with the CALL−IN when it executes and compare it with the value embedded in the RETURN−OUT. Check value mismatches will cause an error and stop execution. This ensures RETURN instructions transfer control flow to after the correct CALL instruction and not to after a different CALL to the same function. The extra instructions ensure secure execution of re-used functions with only two extra instructions per function call. The use of these instructions is demonstrated in Figure 5.

However, some attackers may attempt to avoid the RETURN−OUT check by returning to $x + 2$ or $y + 2$. Therefore, the scheme must protect against attackers skipping CALL−IN or RETURN−OUT instructions. This necessitates an extra check to ensure the instruction executed after a CALL−IN is a CALL and after a RETURN is a RETURN−OUT. This check must also ensure CALL is only executed after CALL−IN, and that RETURN−OUT may only follow a RETURN. Furthermore, CALL−IN not followed by CALL or RETURN not followed by RETURN−OUT also indicates an error. These checks are made using two one-bit processor flags. These flags are CALL−CHECK and RETURN−CHECK and are set by the executing of CALL−IN and RETURN and are reset by CALL and RETURN−OUT. These instructions must always be executed in series, so the check will ensure this by throwing an error if any instruction is executed while a flag is set other than the instruction to reset the flag. Similarly, an error must occur when an instruction to reset a flag is called when the flag is not set.

For example, a CALL−IN instruction executes setting the CALL−CHECK flag and storing the Sec Value. If the next instruction is an ADD instruction or any other instruction other than a CALL,

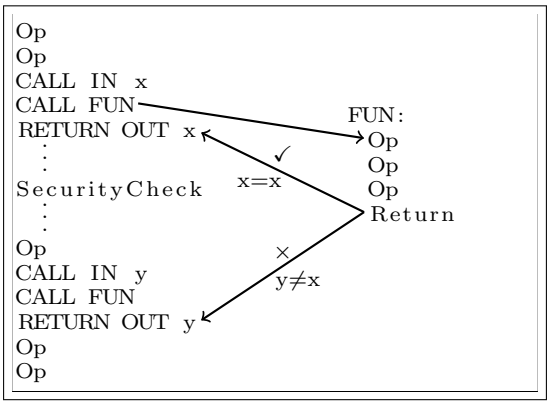

Fig. 5. The extra operations added to the instruction set secure the RETURN after execution of re-used function code.

an error will occur. However, if the next instruction is a CALL, the CALL−CHECK flag will be reset and the processor will execute the function. These additional registers protect the CALL−IN to CALL and RETURN to RETURN−OUT transitions and with the value checks ensure the security of instruction-independent transitions.

However, it is common for functions to call functions. In these cases, a second stack similar to the SCS described in Section 3.2.2 will be used. The check value is added to a stack with the execution of the CALL−IN instruction. The check is made when the RETURN−OUT instruction is executed by comparing the value in the instruction with the top of the Sec Stack. To aid performance, the check value could be stored in a processor register and compared with the instruction check value with no memory access. After the check, the stack is popped to the register at any time before the next RETURN−OUT instruction. This may not help when RETURN and RETURN−OUT instructions are executed in a series of pairs, but may otherwise improve performance.

5 IMPLEMENTATION

This section describes the Secure-Execution Processor (SEP), the proof-of-concept implementation of the scheme proposed in Section 4. The details of the prototype are described in three subsections. Section 5.1 discusses design decisions made when developing the prototype and the hardware used. Section 5.2 describes the assembler developed to compile programs for execution on the SEP. Finally, Section 5.3 covers the hardware of the prototype and how it realises the scheme proposed.

5.1 Design Decisions

The implementation was developed on a Xilinx Spartan-6 FPGA SP601 evaluation board [26]. This board contains a Spartan-6 as well as block RAMs, LEDs, push-buttons, switches, an ethernet port, general-purpose I/O headers and a serial interface. All can be attached to designs loaded into the FPGA [26]. The prototype used block RAM to store software and the serial port and LEDs to demonstrate operation. A high-level system diagram of the prototype is shown in Figure 6.

As well as the hardware developed, a basic application was written to execute on the SEP. This is a simple program written in assembly language to output data to the connected PC via a UART interface[1]. Two tasks were carried out by the application: printing “Hello World!” to the terminal via the UART port and then flashing LEDs based on counter values. In total, the application

[1]VHDL code from the PicoBlaze [25] soft-core processor was used to control the UART interface.

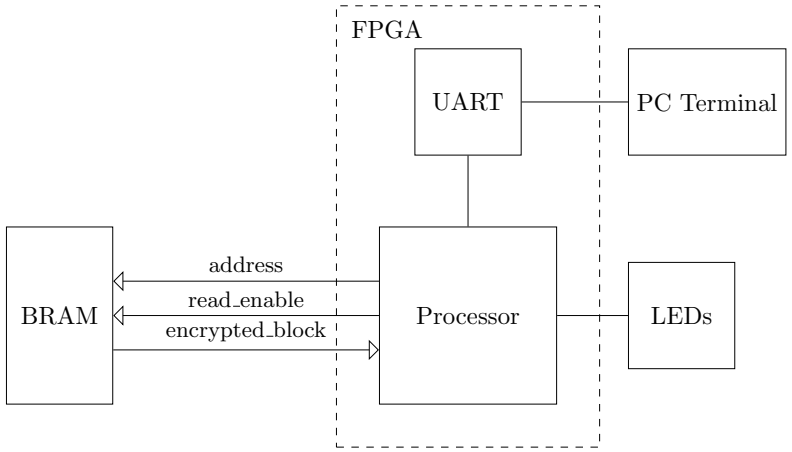

Fig. 6. The prototype included serial and LED outputs to demonstrate correct execution.

(Appendix A) comprised 68 instructions, including multiple `CALL`/`RETURN` pairs and loops using conditional `JUMP` instructions.

5.1.1 Choice of F. Section 4.2 stated the masking function combines the operation, previous masked instruction and location of the operation. However, this idea requires more precise definition before implementation. As suggested by Lee et al., the prototype uses a basic block structure for the application [16]. In the basic block model, an application is split into blocks containing a fixed number of instructions where each block is executed in entirety or not at all. Execution of a basic block always starts with the first instruction and always finishes with the last instruction. `JUMP`s to or from the middle of a block are forbidden. This increases the overall size of the application as `NOP` (no-operation) instructions have to be added to comply with the model. Obviously, extending program length increases execution time. However, it allows the unmasking process more time to complete as once a block is ready there is a period of time based on the block length until the next block is needed. As a result, the prototype did not need to pause execution at any time to perform extra load or unmask instructions, unlike the scheme of Lee et al. [16].

Basic blocks allow more efficient use of the block cipher by matching basic block size to the block size of the block cipher used. However, the function F needs to be explicit for implementation. The function used by Lee et al. in 2016 is not suitable for two reasons. It is reproduced below with a basic block model translation:

$$Op'_x = Op_x \oplus F(Op'_{x-1})$$
$$B'_x = B_x \oplus B(B'_{x-1}),$$

where B_x is the basic block at location x and B'_x is the masked block. The first reason this is not suitable is the value x is absent from the computation, this allows attacks on the scheme as stated in Section 4.2. The second reason is that any bits flipped in B'_x will be flipped in the unmasked block, allowing an attacker to change instructions executed, violating Requirement R3. Therefore a new masking function is needed to protect instructions from modification. The properties needed of the masking function are: inclusion of the previous block, inclusion of the block address and prevention of attackers from making targeted changes to any bits in the unmasked instructions. To achieve these goals, the following functions for masking and unmasking will be used:

Cipher	Block Size	Key Size
AES	128	128
KATAN	32	80
	64	80
PRESENT	64	80
	64	128
PRINCE	64	128
RECTANGLE	64	80
SIMON	32	64
	64	128
SPECK	32	64
	64	128

Table 1. Lightweight block ciphers implemented by Maene and Verbauwhede in [20].

$$B'_x = B'_{x-1} \oplus Enc_{k \oplus x}(B_x)$$
$$B_x = Dec_{k \oplus x}(B'_x \oplus B'_{x-1}).$$

The exclusive-OR before decryption ensures attackers are unable to target a bit or bits to change in the resulting unmasked instructions. As each masking is now dependent on the location of the block, this is due to mixing the address into the encryption key, this does not decrease scheme security as long as the cipher used is secure against related key attacks. Since the masking key is now dependent on the location of the instruction, any rearrangement of basic blocks ensures they are not correctly unmasked. With a strong block cipher, decryption can be considered a pseudorandom function. Therefore an attacker modifying B_x or B'_{x-1} cannot make predictable output changes.

5.1.2 Cipher Used. As discussed previously, the function used to protect instructions needs to produce pseudorandom outputs unpredictable by an attacker. Block ciphers provide the guarantees required and so several were considered for use in securing the application. The list of ciphers considered is included in Table 1, all are considered lightweight apart from AES which is included for reference. However, each cipher has different block sizes, key lengths and complexities. The ciphers in Table 1 were considered because they have single-cycle implementations developed by Maene and Verbauwhede in 2015 [20], that are publicly available via Github [19]. The execution of a basic block may only take a small number of cycles, therefore a single-cycle implementation allows the scheme to be used with less time overheads. This also avoids complicated use of pauses and unpauses such as those encountered by Lee et al. [16].

For the prototype, the PRINCE cipher was used. PRINCE is a lightweight block cipher developed by Borghoff et al. in 2012 [3]. It features a 64 bit block size, 128 bit key size and has been the target of multiple competitions with prizes awarded based on attack strengths [3, 21, 22]. It was designed for efficiency in hardware and was the smallest implementation in the 64/128 class and fastest overall cipher in Maene and Verbauwhede's study [20]. Therefore, due to the high-security, good block size and presence of a published, good, single-cycle implementation, PRINCE was the best choice for use in F.

5.2 Assembler

To assist in preparing applications for the `SEP` by converting instructions into machine code, performing necessary encryptions and enforcing the program structure required, an assembler was developed. The assembler takes as input a program written in `SEP`-assembly and outputs

Opcode	Instruction	Operands
0	LOAD sX, sY	2
1	LOAD sX, kk	2
2	AND sX, kk	2
3	INPUT sX, pp	2
4	ADD sX, kk	2
5	SUB sX, kk	2
6	CALL aaa	1
7	CALL-IN ccc	1
8	JUMP aaa	1
9	JUMP NZ, aaa	1
A	JUMP C, aaa	1
B	RETURN	0
C	RETURN-OUT ccc	1
D	OUTPUT sX, pp	2

Table 2. Only the instructions needed for the test application were implemented and thus some common instructions are missing.

the memory contents to be loaded and executed by the SEP. This section describes the assembly code, instructions implemented and considerations for ensuring a legal program is output from the assembler.

For the SEP, a new assembly language dialect was written. The language developed is based on Picoblaze assembly [25] with most instructions removed - instructions not used by the example application were not implemented leaving a minimal instruction set. This instruction set allowed the opcode of an instruction to require only four bits, granting the SEP a 16-bit instruction set rather than the 18-bit size of PicoBlaze. However, the main benefit of shrinking the instruction size is instructions fit into block sizes of 2^n bits[2]. This enables basic blocks to easily match common block cipher block sizes of 64 or 128 bits, such as that of PRINCE [20]. The complete set of SEP opcodes is listed in Table 2.

As stated, SEP instructions are 16 bits wide. This is split between four bit opcodes and 12-bits for the operand/s. There are five types of operand in SEP instructions: register IDs, data values, IO addresses, memory addresses and security values. Register IDs are 4-bit numbers that select one of the 16 registers available to the SEP Arithmetic and Logic Unit (ALU). Data values are 8-bit numbers used by the ALU, these can be saved, AND-ed, added or subtracted into registers. IO addresses are 8-bit ID numbers of Input or Output devices attached to the SEP, IO addresses select devices to send data to or receive data from one of the ALU registers. Memory addresses are 12-bit addresses used with CALL or JUMP instructions to determine the new value for the PC (subject to a flag check for conditional JUMPs). Finally, security values are 12-bit values used by CALL−IN and RETURN−OUT instructions.

As well as translating human-readable instructions into SEP machine code, the assembler also enforces the basic block structure required. The assembler also adds the required CALL−IN and RETURN−OUT instructions and generated the security values. Therefore, translation and insertion of instructions are the main tasks of the assembler.

To enforce the basic block model there are two main criteria the assembler has to manage. As stated, a basic block always starts with the first instruction and finishes with the final instruction. Therefore any control-flow instruction must meet two criteria: always be at the end of a basic block and always point towards the start of a basic block. These can intersect if one control-flow

[2]Where $n \geq 4$

instruction is the destination of a second control flow instruction. In this case, the first control-flow instruction must be placed at the end of a basic block by inserting NOP instructions until this achieved. Then the second control-flow instruction must be modified to point toward the start of the block containing the first instruction. In most cases however, both criteria do not intersect and in the example program most instructions moved needed to be sent to the start of a basic block or the end.

Assembler program modification can thus be summarised in three points: ensuring JUMP instructions are only at the end of basic blocks, ensuring JUMP destinations are only at the start of blocks and adding CALL−IN and RETURN−OUT instructions where needed.

5.3 Hardware

This section describes the hardware element of the prototype. The hardware developed is a softcore processor, specified in VHDL, for synthesis to a Xilinx Spartan-6 FPGA. Use of an FPGA allows for more rapid prototyping and allows the design to be modified and shared more easily.

The SEP is comprised of six main components: the unmask unit, the data reader, the PC update, the PC/Sec stack, the IO handler and the ALU. Each component in the system completes different tasks and is responsible for updating different signals in the processor. A simplified model of the processor is shown in Figure 7. For clarity, some of the helper signals have not been shown such as internal flags identifying the type of the last instruction in the current block, the zero and carry flags and access to the top of the stacks given to data reader and PC update. We will now describe the components in Figure 7.

The unmask unit receives data from memory and saves it as either an instruction block or helper data block. Use of a received block is calculated based on the current PC value and the state of the execution of the current instruction. The unmask unit also latches helper data used in sequential transitions. The PRINCE cipher component implemented by Maene and Verbauwhede [20] is included in the unmask unit. The cipher decrypts the next block depending on the PC value set during execution of the fourth instruction in the current block. Unmask unit tasks are time critical and follow the schedule in Figure 8.

The data reader sends addresses and enable signals to the memory to request data blocks. BRAM devices on the SP601 require the address and enable signals to be held for one cycle before a memory block is returned. The data reader uses an internal instruction state with the current PC value to time when to set and reset address and read enable signals. Like the unmask unit, data reader tasks are time critical and must follow the execution schedule shown in Figure 8.

The PC update maintains the current Program Counter value and updates it as each instruction is executed. The PC is updated based on the internal clock of the processor and also the current operation executing. Each instruction requires two cycles for execution, the PC is updated after the first cycle to indicate the location of the next instruction. PC update is critical to scheme security as it handles processing instruction transitions.

When a CALL instruction is executed, the PC/Sec stack stores the return address and the last CALL−IN security value. The PC/Sec stack, upon CALL instruction execution, stores the return address and the last CALL−IN security value. Security values from CALL−IN instructions are latched by the PC/Sec stack before they are pushed by the CALL instruction. The PC/Sec Stack displays the top of the stack to the data reader to aid execution of RETURN instructions. Access to the return address allows RETURN instructions to be processed similarly to unconditional JUMP instructions as the next address is known to the data reader.

The IO handler sets the read/write strobe signals and the IO port ID values based on the current operation being executed. When an input or output signal is written or read, the write/read

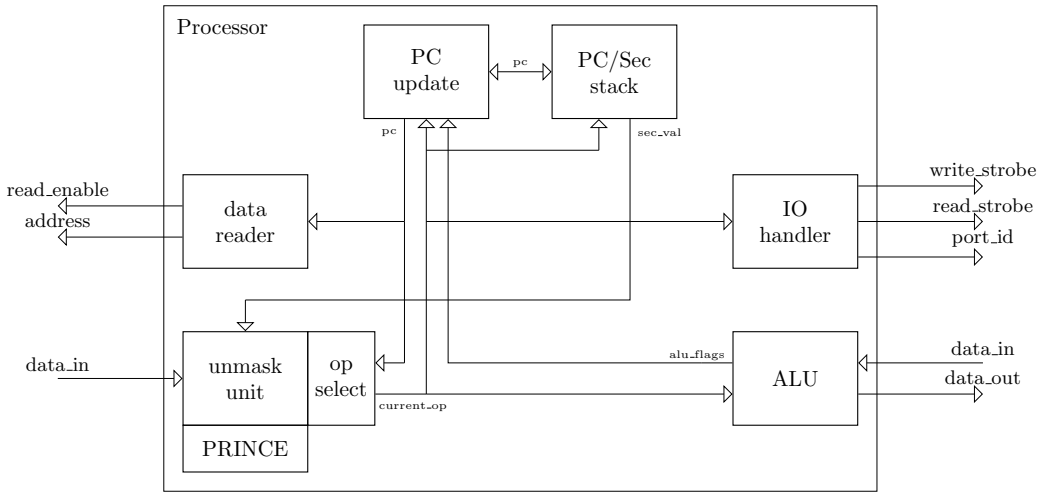

Fig. 7. The Secure-Execution Processor consists of six main elements.

strobe signal is set high for one cycle. This synchronises `SEP` IO reads/writes with the attached input/output devices providing/receiving data.

Finally, the ALU executes arithmetic or logical instructions and maintains the processor registers and zero/carry flags. The ALU also connects registers with IO devices when `INPUT` or `OUTPUT` instructions are executed. As the instruction set (Table 2) does not include a `NOP` instruction, loading a register with itself is used instead as this makes no change to data stored. Specifically, `LOAD s0, s0` is added by the assembler to enforce the basic block model. However, this requires extra ALU logic to maintain program functionality. Normally, when a `LOAD` is executed, the carry flag is reset and the zero flag is only set if the register changed becomes equal to 0. However, this affects `JUMP NZ` instructions if a `LOAD s0, s0` is inserted before the instruction as flags will set/reset according to the value of s0, not last instruction before the `LOAD s0, s0`. Therefore, the ALU was modified to preserve the zero and carry flags if the two registers provided with a `LOAD` instruction are equal. This prevents `NOP/LOAD s0, s0` instructions from effecting program logic.

When executing the application, different transitions between basic blocks have different requirements for the data needed. For example, if the transition from the current block to the next is sequential, the only memory load required is loading the next block in memory. In the sequential case, the helper data for the next block is the current block which is latched to protect sequential transitions. The most demanding case is when the final transition is conditional as this requires the next block, the `JUMP` block and the `JUMP` helper data to all be read by the data reader. As only one memory read can be performed per cycle, the timing of the memory reads for each transition must be met precisely. A table of the data read actions for each type of transition is found in Table 3. Note: each instruction requires two cycles for execution. Changes to the program signals in a simulation of the prototype are shown in Figure 8.

To enforce the security requirements of the proposed scheme, a "kill switch" was added to the design of the processor. Each component shown in Figure 7 is driven by an internal clock, which is defined simply by:

$$int\_clk = clk \& kill\_switch,$$

Op	Cycle	PC	Type of JUMP at end of block		
			Sequential	Unconditional	Conditional
00	0	00			
	1	01			
01	2	01			
	3	10			load B'_x
10	4	10			load B'_{j-1}
	5	11	load B'_x	load B'_{j-1}	load B'_j
11	6	11	load B'_{x+1}	load B'_j	load B'_{x+1}
	7	00	Unmask next block		
$[x+1, j]\|\|00$	8	00	Execute next block		

Table 3. Sequential transitions require less memory accesses than unconditional JUMPs which require less than conditional JUMPs.

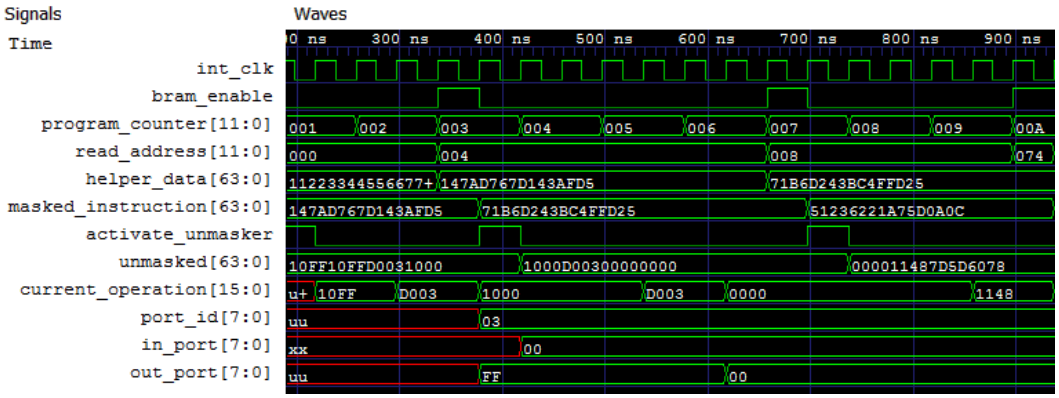

Fig. 8. The Secure-Execution Processor updates each signal according to a strict schedule.

where *kill switch* is initialised as 1 and set to 0 whenever a security problem occurs. We define this security problems as: CALL−IN/RETURN−OUT security value mismatch, CALL without CALL−IN (and vice versa), RETURN without RETURN−OUT (and vice versa).

6 ANALYSIS

This section analyses the scheme proposed and implemented in this work. First, the proposal in Section 4 is checked against the requirements derived in Section 2.4. The second half of this section analyses the implementation described in Section 5.

6.1 Requirements Analysis

Recall that Section 2.4 listed six requirements for the proposed scheme. Requirement R1 is to provide software confidentiality to ensure attackers cannot reproduce the software and violate the software binding. A function F is used to conceal the instructions making up the application executed by the SEP. The F implemented includes PRINCE for encrypting/decrypting blocks of application instructions and ensure software confidentiality.

Requirement R2 needs each application to be bound to just one device. Each device is initialised with a different encryption key, so each masking is unique to a single device. No other device other than the intended device will be able to unmask the blocks and execute the program.

Requirement R3 requires the scheme to prevent attackers making meaningful changes to the application. Section 5.1.1 describes the F used in the implementation and how the use of PRINCE prevents attackers from predicting the consequences of any changes made.

Requirement R4 demands that sequential transitions between instructions be secured. This is achieved by the block chaining in the unmaking functions. The previous executed block is used as helper data without being loaded from memory, this ensures the security of sequential transitions..

Requirement R5 requires that instruction-dependent transitions, such as those following `JUMP` and `CALL` instructions are secured. This is achieved by including the location of the block into the masking function, ensuring the block jumped to cannot execute correctly if it has been modified as each masking is address dependent.

Finally, Requirement R6 states instruction-independent transitions (such as after `RETURN` instructions) must be secured. In the proposed scheme this is met by using `CALL-IN` and `RETURN-OUT` instructions with the PC/Sec Stack to store and check the security values. These instructions must be executed before and after `CALL` and `RETURN` instructions, and the security values must match for execution to continue otherwise the kill switch activates. Instruction-independent transitions are secured by the proposed scheme.

6.2 Implementation Analysis

The prototype implemented the scheme proposed that meets the requirements listed in Section 2.4. This subsection describes the strengths and limitations of the prototype.

The main design decision required when implementing the proposed scheme was to use the basic block model and PRINCE for encrypting the basic blocks. PRINCE was chosen due to the low latency and small size of the single-cycle implementation of Maene and Verbauwhede [20]. The single cycle implementation aided the `SEP` to meet deadlines for instruction readiness. However, as shown in Table 3, for each possible transition there were free cycles available while the block was executing. Therefore, the single cycle implementation is faster than required and a slower implementation could be used if the timing of conditional `JUMP` instructions could be met. Multiple unmask operations while executing the block might be required in this case to avoid the need for pauses. A current advantage of the implementation over other schemes is the avoidance of delay in the execution of instructions.

Cipher choice determined the size of basic blocks to 64 bits/four instructions. This limits the number of `NOP` instructions needed to be inserted to three before a `JUMP`. However, in the case of a `CALL` instruction finishing a block, four additional instructions are needed, as we will need a `CALL-IN` as well as three `NOP`s. Furthermore, the need for control-flow instructions to point towards the start of basic blocks requires even more `NOP` instructions to be added. Due to the extra instructions added, the test application was expanded from containing 68 operations to containing 160 operations after the basic block model conversion and `CALL-IN/RETURN-OUT` instruction addition. However, enforcing the basic block model only extended the application to 108 instructions, the proposed scheme added 52 instructions more than a basic block-only scheme. The example program contained 14 `CALL` instructions, meaning a minimum of 28 instructions needed to be added, and the remaining 24 additions were required to fit the basic block model. Therefore, despite the implementation requiring no pauses in execution, it does still reduce application performance. To achieve the extra security requirements, the length of the program (and thus the number of cycles required to execute) increased substantially.

7 CONCLUSION

In this paper we presented a novel work combining two previously studied problems, Secure Application Execution and Platform Specific Execution. The proposed scheme ensures applications

executing on a device were put there by the manufacturer and are executing correctly from start to finish. The scheme presented is a framework that could be implemented in various ways, and one instantiation was implemented to demonstrate the scheme in practice. A description of the prototype was provided to make clear the steps followed and the constraints present.

Further work in this area would look to different methods for using the `CALL-IN/RETURN-OUT` security values. One potential approach could be to remove the need for a "Sec Stack" and instead use a multiplicative counter if the different Sec Values were prime numbers. Another extension of this work would expand the prototype to a fuller instruction set including storing values in memory. With a full instruction set, an in-depth study of the performance cost of the scheme could be carried out by compiling real-world applications and libraries for execution on the `SEP`.

8 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).

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In Vijay Atluri, Catherine A. Meadows, and Ari Juels, editors, *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, pages 340–353. ACM, 2005.

[2] Mikhail J. Atallah, Eric D. Bryant, John T. Korb, and John R. Rice. Binding software to specific native hardware in a VM environment: The PUF challenge and opportunity. In *Proceedings of the 1st ACM Workshop on Virtual Machine Security*, VMSec '08, pages 45–48, New York, NY, USA, 2008. ACM.

[3] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2012.

[4] Erik Buchanan, Ryan Roemer, Stefan Savage, and Hovav Shacham. Return-oriented programming: Exploitation without code injection. *Black Hat*, 8, 2008.

[5] Nick Christoulakis, George Christou, Elias Athanasopoulos, and Sotiris Ioannidis. HCFI: hardware-enforced control-flow integrity. In Elisa Bertino, Ravi Sandhu, and Alexander Pretschner, editors, *Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy, CODASPY 2016, New Orleans, LA, USA, March 9-11, 2016*, pages 38–49. ACM, 2016.

[6] Louis Columbus. 2017 roundup of internet of things forecasts, 2017.

[7] Lucas Davi, Matthias Hanreich, Debayan Paul, Ahmad-Reza Sadeghi, Patrick Koeberl, Dean Sullivan, Orlando Arias, and Yier Jin. HAFIX: hardware-assisted flow integrity extension. In *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*, pages 74:1–74:6. ACM, 2015.

[8] Ruan de Clercq, Johannes Götzfried, David Übler, Pieter Maene, and Ingrid Verbauwhede. SOFIA: software and control flow integrity architecture. *Computers & Security*, 68:16–35, 2017.

[9] Ruan de Clercq and Ingrid Verbauwhede. A survey of hardware-based control flow integrity (CFI). *CoRR*, abs/1706.07257, 2017.

[10] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Trans. Information Theory*, 29(2):198–207, 1983.

[11] Daniel Genkin, Adi Shamir, and Eran Tromer. Acoustic cryptanalysis. *J. Cryptology*, 30(2):392–443, 2017.

[12] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In Dan Boneh, editor, *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, pages 191–206. USENIX, 2002.

[13] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[14] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[15] R. Krasinski and M. Rosner. Method for binding a software data domain to specific hardware, May 2003.
[16] Robert P. Lee, Konstantinos Markantonakis, and Raja Naeem Akram. Binding hardware and software to prevent firmware modification and device counterfeiting. In Jianying Zhou and Javier Lopez, editors, *Proceedings of the 2nd ACM Workshop on Cyber-Physical System Security, CPSS 2016, Xi'an, China, May 30, 2016*. ACM, 2016.
[17] Robert P. Lee, Konstantinos Markantonakis, and Raja Naeem Akram. Provisioning software with hardware-software binding. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, pages 49:1–49:9. ACM, 2017.
[18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.
[19] Pieter Maene. BlockCiphers, 2017.
[20] Pieter Maene and Ingrid Verbauwhede. Single-cycle implementations of block ciphers. In Tim Güneysu, Gregor Leander, and Amir Moradi, editors, *Lightweight Cryptography for Security and Privacy - 4th International Workshop, LightSec 2015, Bochum, Germany, September 10-11, 2015, Revised Selected Papers*, volume 9542 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2015.
[21] Shahram Rasoolzadeh and Håvard Raddum. Cryptanalysis of PRINCE with minimal data. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2016 - 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings*, volume 9646 of *Lecture Notes in Computer Science*, pages 109–126. Springer, 2016.
[22] Shahram Rasoolzadeh and Håvard Raddum. Faster key recovery attack on round-reduced PRINCE. In Andrey Bogdanov, editor, *Lightweight Cryptography for Security and Privacy - 5th International Workshop, LightSec 2016, Aksaray, Turkey, September 21-22, 2016, Revised Selected Papers*, volume 10098 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2016.
[23] Dean Sullivan, Orlando Arias, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, and Yier Jin. Strategy without tactics: policy-agnostic hardware-enhanced control-flow integrity. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 163:1–163:6. ACM, 2016.
[24] Symantec Corporation. Internet security technical report (istr). Technical report, 03 2018.
[25] Xilinx, Inc. PicoBlaze 8-bit Microcontroller, 2011.
[26] Xilinx Inc. Spartan-6 FPGA SP601 Evaluation Kit, 2018.

A TEST APPLICATION CODE

```
CONSTANT LED_PORT, 02
CONSTANT UART_TX_STATUS, 00
CONSTANT UART_TX_RESET, 03
CONSTANT UART_RX_READ, 01
CONSTANT UART_TX_DATA_IN, 01

start:
  LOAD s0, FF
  LOAD s0, FF
  OUTPUT s0, UART_TX_RESET
  LOAD s0, 00
  LOAD s0, 00
  OUTPUT s0, UART_TX_RESET
  LOAD s0, s0
  LOAD s0, s0
  LOAD s1, "H"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "e"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "l"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "l"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "o"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, " "
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "W"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "o"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "r"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "l"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "d"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, "!"
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, " "
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  LOAD s1, 0D
  CALL waitForUARTFree
  OUTPUT s1, UART_TX_DATA_IN
  JUMP ledStart

waitForUARTFree:
  INPUT s0, UART_TX_STATUS
  AND s0, 04
  JUMP NZ waitForUARTFree
  RETURN

ledStart:
        LOAD s0, 00
loop:   OUTPUT s0, LED_PORT
        LOAD s2, FF
x:      LOAD s1, FF
y:      SUB s1, 01
        JUMP NZ y
        SUB s2, 01
        JUMP NZ x
        ADD s0, 01
        JUMP C reset
back:   JUMP loop
reset:  LOAD s0, 00
        JUMP back
```