TRUST.IO: Protecting Physical Interfaces on Cyber-physical Systems

Chad Spensky*, Aravind Machiry*, Marcel Busch†, Kevin Leach‡, Rick Housley§, Christopher Kruegel*, and Giovanni Vigna*
*UC Santa Barbara †University of Erlangen-Nürnberg ‡University of Michigan §United Technologies
*{cspensky,machiry,chris,vigna}@cs.ucsb.edu †marcel.busch@cs.fau.de ‡kjleach@umich.edu §rick.housley@utc.com

***Abstract*—Cyber-physical systems (CPSes) have been replacing their mechanical counterparts in many safety- and security-critical applications (*e.g.,* door locks, automobiles, and critical infrastructure). However, this paradigm shift has introduced a new software-based attack vector into these historically isolated systems. Since many of these devices are networked, their physical interfaces are vulnerable to both remote and local attackers.**

In this work, we present TRUST.IO, a framework that automatically, and transparently, hardens these physical interfaces against all software-based exploits. More precisely, TRUST.IO ensures that the software on the device cannot access any protected general purpose input/output (GPIO) interfaces unless the command was initiated from a trusted external client (*e.g.,* a key, phone, or centralized server). TRUST.IO exploits the fact that users rarely interact directly with these embedded devices. Instead, users interact with a remote system (*e.g.,* a car key, smart hub, or control system) that ultimately issues commands to the single-purpose embedded device. Thus, TRUST.IO leverages modern embedded processor features to ensure that these critical physical interactions (*e.g.,* actuating motors or reading sensors) will be performed if and only if the command was issued by an authorized external device that can satisfy a cryptographic challenge. We demonstrate that TRUST.IO can be easily applied to existing CPSes, both bare-metal and Linux-based, with minimal runtime overhead and minimal code modifications.

***Index Terms*—trusted execution environment, cyber physical systems, hardware security**

I. INTRODUCTION

The use of cyber-physical systems (CPSes) is becoming commonplace for many day-to-day tasks (*e.g.,* autonomous vehicles, smart homes, robotic assistants, and sensor networks). Moreover, modern infrastructure [1], safety-critical systems [2], and even baby monitors [3] increasingly rely on CPSes for their operation. Unfortunately, the ubiquity of CPSes has brought forth a wealth of new security and privacy issues. As our society develops an increasing dependence on CPSes, there is a critical need to address the growing security threats in this area.

In particular, vulnerabilities that emerge from the physical nature of CPSes can pose substantial risks to life and property. CPS deployments invariably interact with the physical environment by either sensing or actuating. Thus, compromising a CPS device threatens more than data—for example, an attacker who gains control of the physical actuator to an insulin pump could spell disaster for a diabetic user [4]. Indeed, these attacks are no longer theoretical, as numerous real-world attacks have been demonstrated against industrial programmable logic controllers (PLCs) [5], [6], smart home components [7]–[9], and even medical devices [4], [10], [11]. A recent report by Symantec even indicates that most of the United States' power grid was recently infiltrated by a cyber attack that was capable of inducing widespread blackouts [12]. Unfortunately, this trend of exploiting physical interfaces appears to be rising as our physical world becomes increasingly digitized.

The security of these systems ultimately hinges on the ability to ensure that only intended commands are ever actualized by the physical interfaces. Yet, most CPSes lack a method for verifying the origin of a command, which enables attackers to maliciously control these devices by injecting commands remotely (*e.g.,* breaking Zigbee encryption [13]), sending commands locally (*e.g.,* using backdoor credentials [3]), or even compromising the software on the device to actuate the physical interface directly from the memory-mapped region of the peripheral in the system's memory (*e.g.,* a buffer overflow [14]). Moreover, some of these commands will *always* originate from a remote device (*e.g.,* an Internet-connected camera will always be actuated by a remote device), which means that locally running code should, in fact, *never* be allowed to actuate the physical interface directly. Indeed, much of the functionality of these devices, and the logic that controls them, is on a separate device all together, which we call the Client Device. However, the CPS that will eventually perform the requested action currently has no systematic way of verifying the source of the commands that it receives.

To address this growing concern for the security of CPSes, we present TRUST.IO, a system that protects physical interfaces by creating a *trusted path* between the Client Device that issued the command and the CPS that will ultimately act on it. TRUST.IO ensures that only intended commands are executed, even when the communication, underlying operating system (OS), and applications on the CPS have all been completely compromised. It achieves this by leveraging a trusted execution environment (TEE), which are ubiquitous among modern embedded central processing units (CPUs).

TRUST.IO serves as a gatekeeper to these physical interfaces, and any attempt to access these peripherals from the "non-secure world" will automatically and transparently, execute verification code. This verification code will permit the access (*i.e.,* communicating with a peripheral) if and only if a cryptographic challenge has been satisfied by a trusted remote Client Device (*e.g.,* a smart phone that has been paired with the CPS), failing silently otherwise. TRUST.IO also provides the ability to cryptographically verify values that are read from the CPS's sensors, thwarting any attempt to maliciously spoof sensor values to either mask an attack or trigger false alarms. The user experience can remain unmodified (*i.e.,* requiring no more interaction than the unprotected interaction), as the verification can be performed automatically. Indeed, if a TEE is available on the Client Device as well, this interaction can also be hardened against malicious software.

TRUST.IO is designed to augment existing firmware with minimal modifications, in most cases requiring only the implementation of one callback function and the insertion of one function into the system initialization source code of the firmware. Because TRUST.IO is implemented within the secure world (*i.e.,* inside the TEE), the normal world code is able to operate as it did before the addition of TRUST.IO, without any annotations or source code modification for protected memory regions. We also demonstrate the ability to deploy TRUST.IO on Linux-based firmware, providing system-level protection to every application that interacts with any protected peripherals. The implementation of TRUST.IO is completely invisible to the existing applications running on the device, and is compatible with all of the other software-based defenses that have been proposed to harden CPSes. Because TRUST.IO can re-use the same communication channel that is already being used by the CPSes and Client Device for communication, it requires minimal modification to the Client Device as well, leaving the user experience unaltered.

In summary, we claim the following contributions:

- a framework for transparently protecting general purpose input/output (GPIO) interfaces on cyber-physical systems, even in the face of complete software compromise (*e.g.,* the OS and applications);
- a tool for automatically implement these *system-wide* protections, while requiring *no* source code annotations;
- a novel scheme for re-using existing communication channels for a Client Device to verify accesses to a CPS;
- a flexible architecture for securing peripheral input/output (I/O) on TEE-enabled platforms;
- an open-source prototype implementation of our system on a representative embedded ARM platform (https://github.com/ucsb-seclab/trust.io).

II. Background and Motivation

Currently, there are no existing methods for ensuring the security of the physical interfaces of CPSes, aside from standard software-security measures. Many proposed defenses aim to protect embedded devices by performing attestation on their software state (*e.g.,* remote attestation [15] with hardware extensions on the MSP430 processor [16], large-scale swarm attestation [17], and remote control flow attestation [18]). Meanwhile, others have targeted PLCs, employing physics-based intrusion detection mechanisms to attempt to detect stealthy physical interface attacks [19] (*e.g.,* Stuxnet [6]). Yet, none of these solutions are capable of protecting our systems against a runtime, software-based attack.

TEEs have existed for some time, and have continuously evolved; previous examples include Flicker [20], Intel's system management mode (SMM) and recent software guard extensions (SGX). In fact, ARM's TrustZone, which is able to segregate software on the system into *secure* and *non-secure* worlds, is now being included in *all* of their embedded processors. TRUST.IO exploits the proliferation of these TEEs to provide a practical, easily deployed defense against software-based attacks against physical interfaces.

In our ARM-based prototype, TRUST.IO leverages the multiple *worlds* provided by TrustZone and the TrustZone-aware advanced extensible interface (AXI) bus to provide a trusted path between a remote party on a Client Device and the physical interfaces on a critical CPS. These hardware-enforced permissions alleviate the need for attestation of the software, and greatly dampen the potential impact of a software-based vulnerability on the CPS if the system defenses were to fail.

III. Threat Model

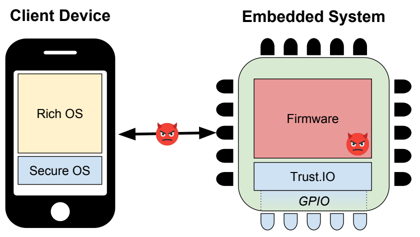
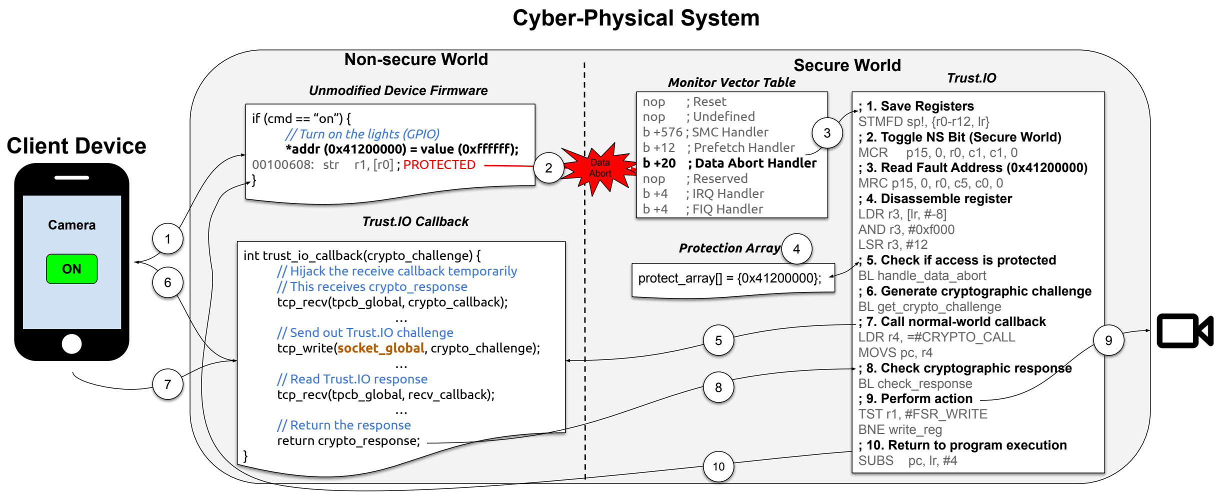


Fig. 1: TRUST.IO threat model, which demonstrates that the GPIO interfaces will remain protected even if an attacker has full control over the communication and the firmware

The goal of TRUST.IO is to mitigate the potential damage caused by a software-based attack against a CPS. TRUST.IO ensures that any malicious software running locally on the CPS *cannot* manipulate the physical GPIO interfaces. This scenario can arise in numerous scenarios, for example: 1) an attacker could use a remote exploit to execute their code on the CPS, 2) an attacker could use a stolen credential to connect to the CPS and issue erroneous commands, or 3) the code on the CPS could have a malicious backdoor to permit unauthorized access. In all of these instances, TRUST.IO can ensure that the GPIO interfaces remain inaccessible, even if the attacker has complete control of the firmware on the CPS. This attacker model can be seen visually in Figure 1.

If the Client Device is also equipped with a TEE, TRUST.IO's implementation on the Client Device could be moved into the TEE to defend against a compromised OS on the Client Device as well (*e.g.,* VButton [21]).

While TRUST.IO can ensure that no unintended physical operations occur, it cannot prevent denial of service (DoS) attacks against the CPS or a reuse of its computational resources (*e.g.,* the Mirai botnet [22]).



```
if (cmd == "on") {
        // Turn on the lights (GPIO)
        *addr (0x41200000) = value (0xffffff);
00100608:   str    r1, [r0] ; PROTECTED
}
```

```
int trust_io_callback(crypto_challenge) {
        // Hijack the receive callback temporarily
        // This receives crypto_response
        tcp_recv(tpcb_global, crypto_callback);
                                ...
        // Send out Trust.IO challenge
        tcp_write(socket_global, crypto_challenge);
                                ...
        // Read Trust.IO response
        tcp_recv(tpcb_global, recv_callback);
                                ...
        // Return the response
        return crypto_response;
}
```

```
nop      ; Reset
nop      ; Undefined
b +576 ; SMC Handler
b +12    ; Prefetch Handler
b +20    ; Data Abort Handler
nop      ; Reserved
b +4     ; IRQ Handler
b +4     ; FIQ Handler
```

```
protect_array[] = {0x41200000};
```

```
; 1. Save Registers
STMFD sp!, {r0-r12, lr}
; 2. Toggle NS Bit (Secure World)
MCR    p15, 0, r0, c1, c1, 0
; 3. Read Fault Address (0x41200000)
MRC p15, 0, r0, c5, c0, 0
; 4. Disassemble register
LDR r3, [lr, #-8]
AND r3, #0xf000
LSR r3, #12
; 5. Check if access is protected
BL handle_data_abort
; 6. Generate cryptographic challenge
BL get_crypto_challenge
; 7. Call normal-world callback
LDR r4, =#CRYPTO_CALL
MOVS pc, r4
; 8. Check cryptographic response
BL check_response
; 9. Perform action
TST r1, #FSR_WRITE
BNE write_reg
; 10. Return to program execution
SUBS    pc, lr, #4
```

Fig. 2: Using TRUST.IO to protect a GPIO write operation on an embedded system that is actuated by a smartphone (*i.e.,* enable recording on an Internet Protocol (IP) camera)

IV. SYSTEM DESIGN

With the increasing prevalence of applications for CPSes, it is neither cost effective nor strategic for companies to postpone the release of a new product, or new feature, to implement stronger security features (*i.e.,* willfully ignoring security to release a product on time). TRUST.IO was designed with these practical commercial needs in mind. TRUST.IO removes the burden of implementing security features from the development phase of these CPS devices, as critical peripheral accesses are protected automatically, and system-wide.

TRUST.IO observes *every* access to *any* peripheral on the system and intelligently permits or rejects each access. TRUST.IO works by conceptually "hooking" accesses to sensitive memory regions on the CPS device it is designed to protect. When a Client Device issues a request to the CPS, TRUST.IO intercepts this resulting I/O on the CPS and issues a cryptographic challenge to the Client Device. This challenge, described in detail in Section IV-D, is constructed in such a way that the Client Device can both (1) verify that the CPS will perform the proper action and (2) respond to the challenge so that the CPS can authorize and execute the original access.

TRUST.IO works in the following way (Figure 2):

1) The Client Device sends a command to the unmodified, untrusted firmware in the non-secure world, which performs the requested actions (*i.e.,* ①).
2) The untrusted firmware then attempts to access the protected peripheral, thereby triggering a *security exception*, which occurs in response to a peripheral access. This exception is then handled by TRUST.IO in the secure world (*i.e.,* ②-③) (Section IV-B2).
3) TRUST.IO then checks its security permissions and, if required, creates a cryptographic challenge that is then sent back to the Client Device, completely unbeknownst to the untrusted firmware (*i.e.,* ④-⑦) (Section IV-D).
4) If the Client Device successfully completes the challenge-response, TRUST.IO transparently performs the requested protected operation and returns control back to the untrusted firmware. Otherwise, it fails silently (*i.e.,* ⑧-⑩) (Section IV-E).

A. Architecture

TRUST.IO was designed to meet the following criteria:

- minimal code modifications to ensure easy adoption and minimize the possibility of programming errors,
- minimal additional code in the compiled binary, since many CPSes have strict space constraints, and
- the ability to reuse the existing communication infrastructure to interact with the Client Device.

Currently, TRUST.IO uses TrustZone and its hardware extensions, which enable peripheral memory regions to be assigned to specific worlds (*i.e.,* secure or non-secure). However, our approach is applicable to any peripheral-aware TEE implementation. We found that TrustZone, while sufficient, was not as straightforward of a solution as one would expect. Thus, we created a method for automatically and transparently moving all of the existing, non-TrustZone-aware, code of an embedded system into the non-secure world, while only restricting access to the protected memory regions.

TRUST.IO's TEE implementation differs from a typical TEE implementation in a few key ways. First, TRUST.IO has no kernel running in the secure world—this is contrary to the model used by popular TEEs that implement a *secure monitor* and depend on the non-secure world to explicitly trigger a world switch. Second, TRUST.IO is never called or interacted with directly by the non-secure world, instead it is only instantiated when a protected memory region is accessed (*i.e.,* a *data abort* exception is triggered). In fact, our approach is completely invisible to the non-secure world code by design. Finally, the secure world invokes a function call back to the non-secure world to receive data from an external entity. This use is novel with respect to most TrustZone applications that make an secure monitor call (SMC) from the non-secure world and pass all of the non-secure data as an argument.

TRUST.IO's low-effort implementation uses the existing First Stage Boot Loader (FSBL) to initialize the board and all of its peripherals[1]. TRUST.IO and the non-secure world are then initialized immediately before the firmware enters its main loop and starts interacting with the user (via the added TRUST.IO function call). The step will automatically move all of the code out of the "secure" world, and modify the permissions appropriately, to execute the firmware in the hardware-segregated "non-secure" world. Similarly, our Linux-based implementation places the entire Linux OS into the non-secure world. The setup function also configures the TEE to protect both TRUST.IO itself and the requested peripheral memory-mapped I/O (MMIO) regions before switching into the non-secure world and continuing the normal firmware execution.

B. Implementation in TrustZone

By default, TrustZone systems boot with all of the configurable memory regions in "secure only" mode, which means that they are inaccessible from the non-secure world. Thus, TRUST.IO first configures *everything* to be accessible from the non-secure world, and then restricts the few memory regions corresponding to the peripherals that are to be protected. All of these peripherals are board-specific, and must be uniquely configured for each deployment. TRUST.IO protects these peripherals by explicitly listing the memory regions that are security-critical, which are verified, and transparently permitting access to all of the non-protected regions of memory.

To protect our verification algorithms and cryptographic keys from being modified by the untrusted, non-secure world code, they are placed in a protected region of memory. This can be accomplished in two ways: 1) using a small region of on-chip memory (OCM) or 2) using a region of random-access memory (RAM) marked as *secure*. By marking the memory that stores TRUST.IO-related code and data as secure-only, any access to TRUST.IO code will go through its own verification algorithm, which is configured to restrict all accesses. Similarly, regions of non-volatile memory that contain TRUST.IO code and data are marked as *secure*, preventing non-secure code from modifying any non-volatile state.

1) Interrupts & Timers: Interrupts are configured in the secure world, thus we must ensure they can still reach code moved to the non-secure world. We configure interrupts in two stages. First, after the FSBL, which executes in the secure world, finishes, every interrupt is marked as non-pending (using the pending clear register, `ICDICPR`). Next, all of the interrupts are configured to be accessible from the non-secure world (using the distributor control register, `ICDDCR`). Then, non-secure interrupts are enabled, by setting secure interrupts to use Fast Interrupt Requests (FIQs), and the *Secure Binary Point Register* is shared between both worlds using the CPU Interface Control Register, `ICCICR`. Upon switching into the non-secure world, the entire secure world configuration is effectively cloned into the non-secure world (*i.e.,* the `ICDICFR0`, `ICDIPR`, `ICDIPTR`, `ICDISER`, and `ICDDCR` registers).

[1]All boards by default boot in the secure world and, unless otherwise configured, all of the code assumes that it is operating in the secure world.

Timers are significantly easier to configure as they are all global. To enable the timers in the non-secure world, TRUST.IO configures the Snoop Control Unit (SCU) register to provide the non-secure world with access to both the private and global timers.

2) External Abort Handler: TrustZone provides numerous configurations for interrupts and exceptions that enable the software to dictate which world (*i.e.,* secure or non-secure) an interrupt is directed to [23]. TRUST.IO modifies the *Secure Configuration Register* to ensure that all *external data aborts* (*i.e.,* an abort thrown by a component outside of the CPU) will be handled in monitor mode, effectively letting TRUST.IO become a person-in-the-middle for all accesses to off-chip peripherals. It is this abort handler that implements all of the logic for TRUST.IO.

While there are numerous types of external aborts, TRUST.IO only handles *precise* external data aborts (*i.e.,* an abort that occurs during an instruction prefetch) that are caused by a denied, non-secure access. In the case of an external abort, only three pieces of information are provided: 1) the Data Fault Status Register (DFSR), which contains information about the type of abort; 2) a banked *link register*, which can be used to return from the abort and to infer the instruction that caused the abort; and 3) the Data Fault Address Register (DFAR), which contains the address that caused the fault (*i.e.,* the protected memory address). TRUST.IO uses these three registers to both transparently execute non-protected accesses and enforce a cryptographic challenge-response protocol with the Client Device for protected memory regions. In our experience, *all* of our observed aborts caused by access to protect MMIO were *precise*, and thus could be handled. Others have shown that memory can always be configured to force precise data aborts in the event that imprecise data aborts are thrown [24].

To execute the requested operation (*e.g.,* write a value to memory), TRUST.IO must first disassemble the instruction that caused the abort. The location of the instruction responsible for the abort is pushed onto the secure stack. Necessarily, the instruction must be either a `str`, `ldr`, `stc`, `ldc`, or `swp`, as these are the only instructions that can be used to interact with memory in ARM. Similarly, the DFSR contains information on whether the instruction was a read or a write. Thus, TRUST.IO is able to accurately disassemble which register that was the source of a write or the destination of a read, and perform the appropriate action with the source or destination address provided in the DFAR.

C. Remote Callback Function

The remote callback function is critical to TRUST.IO, but it need not be secure. The callback's sole purpose is to facilitate a communication channel with the Client Device to send and receive encrypted data. While it may make sense, in some cases, to implement a standalone program as the callback function, we believe it is easier, and more convenient, to simply reuse the existing channel (*e.g.,* an IP protocol) that is already open with the Client Device. In almost every case, this channel must exist because the Client Device must have issued an initial command to trigger the physical action.

D. Cryptographic Implementation

For our prototype, we implemented the minimal protocol that would permit us to accurately assess the performance impacts of our system, but do not claim that is either secure or optimal. The protocol uses pre-shared 128 bit symmetric keys and encryption is done using Advanced Encryption Standard (AES) [25] in cipher block chaining (CBC) mode. The ciphertext includes the address being accessed (*i.e.,* DFAR); DFSR, which indicates whether a read or write was requested; the value to be written or the value that was read; a nonce; and a counter. The nonce is obtained from a global timer on the system in our implementation; however, a hardware-based random number generator would be preferred in practice. The combination of the counter and nonce ensures that the plaintext will never repeat, with the nonce providing a unique value for each challenge-response interaction and the counter ensuring that no messages can be replayed. Because we ensure that our plaintext will never repeat, we simply use a static initialization vector (IV) when initializing our AES cipher.

1) Key Management: We also implemented a simple key update procedure, which would be required to adequately diversify CPSes in practice. As implemented, our plaintext challenges could potentially repeat after the counter is exhausted, thus requiring the key to be periodically updated to ensure long-term confidentiality.

Because of the limited user interface (UI) on these devices, we do not require the developers to implement any special functionality for updating keys. Instead, we simply trigger the pairing procedure inside of any protected access, based on the detection of a physical input (*e.g.,* a button). For example, to update the key on a smart bulb, the user would hold a button on the light bulb, connect their Client Device to the light, and issue the light "on" or "off" command. While reading this physical interface during every interaction incurs some overhead, the general approach ensures, again, that developers need not worry about any security-related implementation details. When the data abort handler is triggered, the button will trigger a key update procedure in addition to the normal access-control protocol. The Client Device would then update the key, which is a symmetric key in our example, and would be able to use this key for all subsequent accesses. Reusing existing interactions not only satisfies our initial requirement of minimal code modification, but is also practical for CPSes, which inherently have limited user interfaces and methods of interaction. Moreover, since the device initializes the key exchange protocol, this enables the device to automatically force a key update once it has exhausted its counter.

E. Client Implementation

The user experience on the Client Device *does not change at all*, since all of the cryptographic verification is done in the background, unbeknownst to the human user. This is done by augmenting any command in the client that interacts with protected functionality on the CPS with TRUST.IO-aware versions, which is made possible by a library in the case of our prototype implementation. Because each command (*e.g.,* unlock door) is performed in a stateful way (*i.e.,* the command is sent, the challenge is received, and the response is sent), there is no risk of the Client Device blindly responding to an unsolicited challenge. Moreover, these augmented client commands also check the freshness (*i.e.,* that the request is not a replay of an old challenge), the MMIO address, and functionality (*i.e.,* whether the CPS is reading or writing to the value) of the challenge automatically, to ensure that no malicious functionality on the CPS is trying to modify the action or replay a previous challenge. The stateful protocol and encryption ensure that the Client will approve the challenge if and only if it issued the initial command.

1) Semantic Gap: TRUST.IO introduces an inherit *semantic gap* where the enforcement of the policy is at a lower level of abstraction than that actual of the command it seeks to protect. For example, with the simple case of unlocking a door, we actually want to protect the "unlock" command, but instead TRUST.IO will see a memory address and the value that is being written to it. Maintaining this correspondence is quite tractable in practice. Indeed, the developer could hardcode these values. Even easier however, these expected values could be automatically inferred by training the Client Device on the TRUST.IO-enabled device to infer which low-level operations are induced by each high-level command. For example, if TRUST.IO were placed in "learning" mode in light bulb example, the user could simply turn the light *on* and *off* and the Client code would automatically infer that *on* should write `0x1` to `0x41200000` and the *off* should write `0x0` to the same address. These parameters would then be stored permanently, and every subsequent command would be protected by TRUST.IO.

V. EVALUATION

We implemented two indicative scenarios: a bare-metal application that actuates a light-emitting diode (LED) (Section V-A), and a Linux-based firmware that incorporates TRUST.IO into existing drivers and applications (Section V-B). In both of these instances, we built upon open-source implementations and initialized system with a pre-shared 128 bit AES key and static IV.

For our complete, bare-metal implementation we used a Xilinx ZC702 board, which is equipped with a dual core Cortex-A9 processor, operating at 333 MHz, with the TrustZone extensions enabled. While we would have preferred a TrustZone-enabled Cortex-M development board as well, we were unable to obtain an appropriate development board. Nevertheless, by implementing TRUST.IO on the more complex A-series processors, we demonstrate the generalizability of our approach to all CPSes. Our Linux-based implementation was done on a 1.2 GHz HiKey development board, which is supported by the Open Portable Trusted Execution Environment (OP-TEE) [26], a popular open-source TEE.

A. Bare-metal Networking Application

For our bare-metal evaluation, we used a stand-alone network "echo" program that uses the popular Light-Weight IP (LWIP) library, which opens a Transmission Control Protocol (TCP) server and echoes back any data sent to it. We modified it to accept three ASCII commands: "on," "off," and "read," which will turn an LED on or off, or read the value stored in the protected GPIO memory region. The state of the LED is manipulated through MMIO, such that writing a 1 or 0 in that memory location would turn the light on or off, respectively. This kind of memory-mapped interaction is indicative of various workflows that are common to embedded systems (*e.g.,* actuating a relay or transistor, controlling a motor, or interacting with a peripheral device).

To enable TRUST.IO protections on this application, we first added the MMIO address (`0x4120000`) to our list of protected memory regions, and added our setup and world switching function call right before the `start_application()` function call in the main file of the source code. Thus, ensuring that the firmware boots exactly as it did before, but with the user-facing code isolated in the non-secure world, such that any attempt to access the LED will be protected by TRUST.IO.

a) Callback function: To incorporate our callback function with the existing TCP connection, we made two parameters global to use their state in our own callback function: (1) the TCP protocol control block structure, which contains all of the state information associated with the TCP connection, and (2) a pointer to the receiver callback function that is triggered when any data is received. Our callback first uses the TCP protocol control block structure to send the challenge, and then temporarily replaces the callback with our own simplified receive function to obtain the response to the cryptographic challenge. After the response is received, our callback restores the TCP receive function and triggers the SMC instruction, passing the returned cryptographic challenge back to our secure world code, which verifies the response. This entire callback function was implemented in *46* lines of `C` code.

1) Overhead: TRUST.IO is not only easy to add to an existing bare-metal application, but also has low system overhead, both in terms of runtime and code size. Our implementation was written almost entirely in ARM assembly to ensure minimal overhead. First, we compiled the Executable and Linkable Format (ELF) file of the binary with optimizations (gcc `-O1`), with and without TRUST.IO enabled. Table I shows that TRUST.IO adds only $\approx 0.1\%$ to the total binary size, with a majority of that increase attributed to the 2,468 bytes, or 617 instructions, added to the `.text` segment. We consider this a small increase in code size to gain the security guarantees provided by TRUST.IO.

This increase does *not* include a cryptographic library within the secure world. With our AES library included, the total binary size still only increased by $\approx 0.7\%$, with a majority of that, again, stemming from the `.text` segment. In this instance however, the `.data` segment also increased by 37%, as we must allocate more memory for the keys and cryptographic state. While the embedded components of CPSes typically have stringent space constraints, we believe that a worst-case, 16 Kilobyte overhead is a reasonable trade-off in most scenarios.

We also evaluated the timing overhead incurred by TRUST.IO. This was done by recording the end-to-end time taken to turn the light on and off as observed from a Client Device, both with and without TRUST.IO. The LED was then cycled on and off 1,000 times, resulting in 2,000 total observations. The unmodified bare-metal server, without TRUST.IO, yielded a round-trip time of $0.615(\pm 0.082)$ ms. With TRUST.IO enabled, we saw a significant increase in time, now $204(\pm 0.63)$ ms. However, this time is still well-below empirically measured human response time [27], leaving the end user experience unaltered. Additionally, we recorded the individual operations associated with TRUST.IO, which show that almost all of this increased time is spent in the non-secure world communicating with the Client Device, and not within the core TRUST.IO code. In fact, Table II shows that almost all of the time (≈ 200 ms) is spent interacting with the Client Device over the network, with less than 0.4 ms spent switching between worlds and enforcing access control. Note that all but 0.001 ms of time in the secure world is spent generating and verifying our cryptographic challenge (*i.e.,* 0.043 ms to generate the challenge and 0.35 ms to verify it). The rest of the TRUST.IO code is thus only executing for 0.002 ms on both protected writes and reads. While we are confident that some engineering effort could speed up Ethernet interactions and key generation, we find these results

TABLE I: Compiled binary size comparison of our bare-metal prototype without TRUST.IO, with TRUST.IO, and with TRUST.IO and an AES library

	Total	`.text`	`.data`	`.bss`
Unmodified	2,344,620	122,500	3,832	2,218,288
Δ T.IO	3,072 (0.13%)	2,424 (2.0%)	648 (14.5%)	–
Δ T.IO+AES	16,436 (0.70%)	15,020 (12.3%)	1,416 (37.0%)	–

TABLE II: Fine-grained timing of TRUST.IO implemented on a bare-metal networking application running a 333 MHz processor. The timings show how long it takes to intercept the access and construct a cryptographic challenge in the secure world (*S: Data Abort*), send the challenge to the Client Device and receive a response in the non-secure world (*NS: Callback*), and verify the response and perform the requested action (*S: Verify/Return*), over 2,000 observations.

Command	S: Data Abort	NS: Callback	S: Verify/Return
write	0.043 ± 0.0001 ms	203.8 ± 0.74 ms	0.35 ± 0.0002 ms
read	0.043 ± 0.0001 ms	201.4 ± 1.49 ms	0.35 ± 0.0002 ms

very encouraging, as the bottleneck is the user-implemented callback and not the core TRUST.IO code.

For memory regions that are not protected by TRUST.IO (*i.e.,* unprotected external memory regions), the overhead incurred by TRUST.IO acting as a passive person-in-middle is less than 2 ns for both reads and writes to the memory-mapped memory.

B. Linux-based Firmware

We demonstrate the generality of our system by extending TRUST.IO to support Linux-based systems as well. To evaluate the feasibility on Linux-based embedded systems, we implemented TRUST.IO on OP-TEE, which provides various cryptographic functions that simplified our implementation. We selected a minimal version of Debian [28] as our untrusted OS, which required minimal additions to add TRUST.IO support. In fact, we only needed to add 119 lines of `C` code overall, with a majority of those executing in the secure world.

We implemented our prototype on the HiKey development board, which has a 1.2 GHz processor. On the HiKey board, peripherals can be configured to be secure-only by writing to the corresponding bit in the TrustZone Peripheral Protection Controller (TZPC) register; unfortunately, their TZPC implementation is proprietary. Thus, we were unable to set the exact bit that needs to be set to configure a specific GPIO pin to be secure-only [29].

Nonetheless, we implemented TRUST.IO by modifying the GPIO driver to make an `smc` call when a TRUST.IO-protected GPIO pin is accessed. Thus, all of the proper TRUST.IO actions are taken, allowing us to accurately measure the incurred overhead. However, without access to the TZPC, an attacker with root privileges could subvert this prototype by writing to the memory directly. In a real-world deployment, where the TZPC configuration is known, we simply need to add a few lines of code to the initialization of OP-TEE to enable the hardware-enforced memory protections (*i.e.,* `SLAVE_PROTX_SET` and `GPIO_SEC_ONLY`).

To evaluate the performance of TRUST.IO on Linux-based CPSes, we implemented a networking application that is similar to the LED example on the bare-metal system (*i.e.,* it accepts commands and actuates an on-board LED). The round-trip overhead of reading and writing to a GPIO pin using TRUST.IO is $5.399(\pm0.413)$ ms and $5.562(\pm0.401)$ ms, respectively. Table III shows the fine-grained timings of the various components. All measurements are averaged over 2,000 observations. Similar to the bare-metal implementation, most of the time is spent in the non-secure world callback interacting with the Client Device.

TABLE III: Fine-grained timing of our TRUST.IO implemented on a Debian based OP-TEE running on HiKey board with a 1.2 GHz processor, over 2,000 observations

Command	Secure Protection	NS Callback	Secure Verify/Return
write	0.278 ± 0.001 ms	4.115 ± 0.184 ms	0.173 ± 0.007 ms
read	0.287 ± 0.006 ms	4.450 ± 0.170 ms	0.176 ± 0.004 ms

VI. LIMITATIONS

TRUST.IO has a few fundamental limitations. First, it requires a TrustZone-enabled processor. Because of the additional round-trip time, TRUST.IO may not be appropriate in certain time-critical applications (*e.g.,* anti-lock brakes). Our experiments demonstrate that a TRUST.IO-enabled system with 333 MHz processor can reply to periodic sensor readings or actuation at a rate of up to ≈ 5 commands per second.

Infinite control loops (*e.g.,* do X every Y seconds), which are common on CPSes are currently not supported by TRUST.IO. If the Client Device is always present, it could in fact continuously authenticate these actions, ensuring proper operation of the device. If the Client Device is removed however, TRUST.IO would need to be extended to not only protect the I/O operation but to encapsulate the entire loop to ensure that no deviation from the *approved* action in the loop could ever be taken. However, in many cases (*e.g.,* an IP camera) this limitation does not apply, as the initial activation of the action could be guarded with TRUST.IO (*i.e.,* turn on the camera), while the subsequent loop could be left unprotected (*e.g.,* reading image frames out of memory).

VII. RELATED WORK

Control Flow Integrity (CFI) [30]: ECFI [31] was the first system that tried to enforce CFI for PLC firmware using compile-time instrumentation. These techniques, although efficient, are highly customized to PLC-based firmware, and their performance on other Linux-based firmware is unclear. mShield [32] uses a combination of runtime instrumentation and selective system call hooking to enforce relaxed backward edge CFI on commercial off-the-shelf binaries running on Linux-based OSes. However, it is still susceptible to kernel exploits and needs operating system support. MProsper [33] uses a hypervisor to prevent code injection attacks by enforcing the W$\oplus$X property on guest OS code pages. However, W$\oplus$X is a weak guarantee and cannot prevent classic code-reuse attacks [34], [35]. Orpheus [36] uses anomaly detection techniques to identify data-oriented attacks. However, this technique only works on embedded systems which run a Linux-like OS. In addition to the relative weaknesses, all of these techniques have a prohibitive performance impact. IBMAC [37] tries to improve performance by using custom hardware for an efficient implementation of shadow stack. However, the required hardware modifications are intrusive and could affect the overall stability of the device.

Software attestation [38], [39]: Swatt [40] provides an attestation mechanism on memory contents, without hardware support. This technique is improved by ASSW [41], by using stride access instead of complete memory access. However, these techniques expect to have a trusted software component on the device, which is hard to enforce without a trusted or isolated execution environment. SEDA [17] provides an approach to perform remote attestation without hardware support. However, while SEDA can attest the validity of the software on the system, it cannot defend against illegitimate commands, leaked credentials, modified sensor readings, or hard-coded backdoors. Unlike CFI and software attestation, which try to prevent software compromise, TRUST.IO aims to protect the device in spite of a complete software compromise.

TrustZone-based protections [42]: ARM TrustZone provides an isolated privileged execution environment, and has been used to implement various features [43]–[46]. C-Flat [18] is one the first techniques that use ARM TrustZone to provide control-flow attestation. C-Flat expects a prior control-flow model to be available for all binaries, which might not be possible for proprietary and legacy binaries. VButton [21] is another system which leverages ARM TrustZone to attest user-driven actions. However, unlike TRUST.IO, this is not transparent, as it requires modifications to the entire system stack (*i.e.,* the untrusted app, the untrusted OS, the TEE, and the server side implementation of the corresponding application). SKEE [47] uses TZ-based page table protections to enforce kernel integrity, however extending this technique to bare-metal embedded systems is not trivial.

Peripheral access control: EPOXY [48] is the first system that tries to use a software solution based on Privilege Overlays to protect the access to physical interfaces. This system is based on an LLVM-based embedded compiler and has source code restrictions. In contrast, TRUST.IO is easily applied to existing bare-metal firmware without additional instrumentation or restrictions. Furthermore, TRUST.IO signs its responses, protecting from peripheral person-in-the-middle attacks [49].

SeCloak [24], which is a TrustZone-based system designed to provide a fine-grained peripheral control on modern smart phones. This system used a custom secure kernel which mediates the access to all peripherals using abort handlers, similar to TRUST.IO. The user can set access control restrictions to the desired peripherals using an untrusted user mode app. This configuration is then confirm with the user using a trusted interface from the secure world. Indeed, SeCloak also uses data abort exceptions in the same way that TRUST.IO does to enforce its restrictions. However, the goals of the two systems are quite distinct. SeCloak aims to provide users with a trustworthy way to restrict access to peripherals on the same device that they are using (*i.e.,* their smartphone), while TRUST.IO ensures that only accesses from vetted Client Devices are permitted, thwarting malware and unsolicited commands. SeCloak also requires the *device tree* file for configuration, which is not available in case of the bare-metal systems. Finally, it requires a custom secure kernel, which makes SeCloak inapplicable to existing secure kernels and impractical on bare-metal systems.

TABLE IV: Comparison table of TRUST.IO alternatives

Solution	Bare Metal	No Custom TEE	Minimal Modification	No Source Restriction	Transparent
VButton [21]	✗	✗	✗	✗	✗
EPOXY [48]	✓	✓	✓	✗	✗
SeCloak [24]	✗	✗	✓	✓	✓
TRUST.IO	✓	✓	✓	✓	✓

VIII. CONCLUSION

As networked CPSes continue to replace traditionally analog systems (*e.g.,* smart homes, automobiles, and components of the critical infrastructure), the need to secure their physical interfaces is of paramount importance. To address this growing concern, we present TRUST.IO, a framework that is capable of protecting the physical interfaces of CPSes in a transparent way, even in cases where the firmware is completely compromised at runtime. TRUST.IO leverages the TEEs (*e.g.,* TrustZone) available on modern processors, to cryptographically verify any access to protected peripherals on a CPS device with a trusted Client Device (*e.g.,* a car key, smartphone, or centralized controller). TRUST.IO provides cryptographic assurance that the command was initiated, and validated by the Client Device, preventing any malicious software on the CPS itself from accessing the interface directly or spoofing read values. We demonstrate that TRUST.IO is practical by providing prototypes for both bare-metal and Linux-based firmware. Both implementations required minimal modifications to their respective source, and had a minimal runtime overhead of $\approx 200\ ms$ on our slower bare-metal system and of less than $10\ ms$ on the faster Linux-based environment. TRUST.IO provides a necessary additional layer of defense to help mitigate attacks against these critical physical interfaces.

ACKNOWLEDGEMENTS

This paper is dedicated to Kevin Leach and Yu Huang, married on May 6th, 2017. Congratulations, and may the future hold nothing but happiness for you both. This material is based upon work supported by the Office of Naval Research (ONR) (Award No. N00014-17-1-2011) and by the Department of Homeland Security (DHS) (Award No. FA8750-19-2-0005). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the ONR or DHS. This was work was also supported by the IBM PhD Fellowship that was awarded to Chad Spensky.

REFERENCES

[1] Y. Mo, T. H.-J. Kim, K. Brancik, D. Dickinson, H. Lee, A. Perrig, and B. Sinopoli, "Cyber–physical Security of a Smart Grid Infrastructure," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 195–209, 2012.

[2] I. Lee, O. Sokolsky, S. Chen, J. Hatcliff, E. Jee, B. Kim, A. King, M. Mullen-Fortino, S. Park, A. Roederer *et al.*, "Challenges and Research Directions in Medical Cyber-Physical Systems," *Proceedings of the IEEE*, vol. 100, no. 1, pp. 75–90, 2012.

[3] M. Stanislav and T. Beardsley, "Hacking iot: A case study on baby monitor exposures and vulnerabilities," *Rapid7 Report*, 2015.

[4] J. Radcliffe, "Hacking Medical Devices for fun and insulin: Breaking the human SCADA systemun and Insulin: Breaking the Human SCADA System," in *BlackHat*, vol. 2011, 2011.

[5] A. Abbasi and M. Hashemi, "Ghost in the PLC: Designing an Undetectable Programmable Logic Controller Rootkit via Pin Control Attack," *BlackHat Europe*, 2016.

[6] N. Falliere, L. O. Murchu, and E. Chien, "W32. Stuxnet Dossier," *White paper, Symantec Corp., Security Response*, vol. 5, no. 6, 2011.

[7] J. Hall and B. Ramsey, "Breaking Bulbs Briskly by Bogus Broadcast," in *Shmoocon*, 2016.

[8] E. Ronen, C. O'Flynn, A. Shamir, and A.-O. Weingarten, "IoT Goes Nuclear: Creating a ZigBee Chain Reaction," http://iotworm.eyalro.net/iotworm.pdf, 2016.

[9] Jmaxxz, "Backdooring the Frontdoor," in *Defcon*, 2016. [Online]. Available: https://media.defcon.org/DEF%20CON%2024/DEF%20CON%2024%20presentations/DEFCON-24-Jmaxxz-Backdooring-the-Frontdoor.pdf

[10] S. Gayou, "Remote Code Execution on the Smiths Medical Medfusion 4000," https://github.com/sgayou/medfusion-4000-research/blob/master/doc/README.md, January 2018.

[11] G. Wassermann, "NXP Semiconductors MQX RTOS Contain Multiple Vulnerabilities," https://www.kb.cert.org/vuls/id/590639, October 2017.

[12] A. Greenberg, "Hackers Gain Direct Access to US Power Grid Controls," https://www.wired.com/story/hackers-gain-switch-flipping-access-to-us-power-systems/, September 2017.

[13] M. B. Barcena and C. Wueest, "Insecurity in the internet of things," *Security Response, Symantec*, 2015.

[14] "CVE-2017-9765: Devil's Ivy," https://nvd.nist.gov/vuln/detail/CVE-2017-9765, 07 2017.

[15] X. Carpent, K. Eldefrawy, N. Rattanavipanon, A.-R. Sadeghi, and G. Tsudik, "Reconciling remote attestation and safety-critical operation on simple iot devices," in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[16] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base," in *Proceedings of the USENIX Security Symposium*, 2013, pp. 479–494.

[17] N. Asokan, F. Brasser, A. Ibrahim, A.-R. Sadeghi, M. Schunter, G. Tsudik, and C. Wachsmann, "SEDA: Scalable Embedded Device Attestation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 964–975.

[18] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 743–754.

[19] D. I. Urbina, J. A. Giraldo, A. A. Cardenas, N. O. Tippenhauer, J. Valente, M. Faisal, J. Ruths, R. Candell, and H. Sandberg, "Limiting the Impact of Stealthy Attacks on Industrial Control Systems," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1092–1105.

[20] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 315–328.

[21] W. Li, S. Luo, Z. Sun, Y. Xia, L. Lu, H. Chen, B. Zang, and H. Guan, "Vbutton: Practical attestation of user-driven operations in mobile apps," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2018, pp. 28–40.

[22] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1093–1110.

[23] R. Lynx, "How to Use ARM's Data-abort Exception," *Embedded Systems Design*, vol. 19, no. 8, p. 46, 2006.

[24] M. Lentz, R. Sen, P. Druschel, and B. Bhattacharjee, "Secloak: Arm trustzone-based mobile peripheral control," 2018.

[25] kokke, "Tiny aes in c," https://github.com/kokke/tiny-AES-c, March 2018.

[26] "Open Portable Trusted Execution Environment," http://www.op-tee.org.

[27] K. Amano, N. Goda, S. Nishida, Y. Ejima, T. Takeda, and Y. Ohtani, "Estimation of the timing of human visual perception from magnetoencephalography," *Journal of Neuroscience*, vol. 26, no. 15, pp. 3981–3991, 2006.

[28] STMicroelectronics and Linaro Security Working Group, "Open Source TEE with Debian," https://github.com/OP-TEE/build/blob/master/docs/hikey.md.

[29] ——, "How do I control GPIO," https://github.com/OP-TEE/optee_os/issues/1065.

[30] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and communications security*. ACM, 2005, pp. 340–353.

[31] A. Abbasi, T. Holz, E. Zambon, and S. Etalle, "ECFI: Asynchronous Control Flow Integrity for Programmable Logic Controllers," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 437–448.

[32] A. Abbasi, J. Wetzels, W. Bokslag, E. Zambon, and S. Etalle, "m Shield - Configurable Code-Reuse Attacks Mitigation For Embedded Systems," in *NSS*, 2017.

[33] H. Chfouka, H. Nemati, R. Guanciale, M. Dam, and P. Ekdahl, "Trustworthy Prevention of Code Injection in Linux on Embedded Devices," in *European Symposium on Research in Computer Security*. Springer, 2015, pp. 90–107.

[34] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented pPogramming: Systems, Languages, and Applications," *Proceedings of the ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, p. 2, 2012.

[35] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," in *Proceedings of the USENIX Security Symposium*, 2015, pp. 161–176.

[36] L. Cheng, K. Tian, and D. D. Yao, "Orpheus: Enforcing Cyber-physical Execution Semantics to Defend Against Data-Oriented Attacks," in *Proceedings of the 33rd Annual Computer Security Applications Conference*. ACM, 2017, pp. 315–326.

[37] A. Francillon, D. Perito, and C. Castelluccia, "Defending Embedded Systems Against Control Flow Attacks," in *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, ser. SecuCode '09. New York, NY, USA: ACM, 2009, pp. 19–26. [Online]. Available: http://doi.acm.org/10.1145/1655077.1655083

[38] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "A Security Framework for the Analysis and Design of Software Attestation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer Communications Security*, ser. CCS. ACM, 2013.

[39] J. Valente, C. Barreto, and A. A. Cárdenas, "Cyber-Physical Systems Attestation," in *Proceedings of the 2014 IEEE International Conference on Distributed Computing in Sensor Systems*, ser. DCOSS '14. Washington, DC, USA: IEEE Computer Society, 2014.

[40] A. Seshadri, A. Perrig, L. Van Doorn, and P. Khosla, "Swatt: Software-based Attestation for Embedded Devices," in *Proceedings of the 2004 IEEE Symposium on Security and Privacy*. IEEE, 2004, pp. 272–282.

[41] B. Chen, X. Dong, G. Bai, S. Jauhar, and Y. Cheng, "Secure and Efficient Software-based Attestation for Industrial Control Devices with ARM Processors," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC 2017. New York, NY, USA: ACM, 2017.

[42] ARM, "ARM TrustZone," http://www.arm.com/products/processors/technologies/trustzone/index.php, 2015.

[43] J. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang, "SeCReT: Secure Channel between Rich Execution Environment and Trusted Execution Environment," *Proceedings of Network and Distributed Systems Symposium*, 2015.

[44] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "TrustShadow: Secure Execution of Unmodified Applications with ARM TrustZone," *arXiv preprint arXiv:1704.05600*, 2017.

[45] S. Zhao, Q. Zhang, G. Hu, Y. Qin, and D. Feng, "Providing Root of Trust for ARM Trustzone Using On-chip SRAM," in *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*. ACM, 2014, pp. 25–36.

[46] C. Marforio, N. Karapanos, C. Soriente, K. Kostiainen, and S. Capkun, "Smartphones as Practical and Secure Location Verification Tokens for Payments," in *Proceedings of the Network and Distributed System Security*, 2014.

[47] A. M. Azab, K. Swidowski, R. Bhutkar, J. Ma, W. Shen, R. Wang, and P. Ning, "SKEE: A Lightweight Secure Kernel-level Execution Environment for ARM," in *Proceedings of the Network and Distributed System Security*. Internet Society, 2016.

[48] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting Bare-metal Embedded Systems With Privilege Overlays," *Proeceedings of the IEEE Security and Privacy Symposium*, 2017.

[49] P. Stewin, "A Primitive for Revealing Stealthy Peripheral-based Attacks on the Computing Platform's Main Memory," in *Proceedings of the International Workshop on Recent Advances in Intrusion Detection*. Springer, 2013, pp. 1–20.