Hardware-assisted Remote Runtime Attestation for Critical Embedded Systems

Munir Geden
Department of Computer Science
University of Oxford, UK
munir.geden@cs.ox.ac.uk

Kasper Rasmussen
Department of Computer Science
University of Oxford, UK
kasper.rasmussen@cs.ox.ac.uk

***Abstract*—Remote attestation, as a challenge-response protocol, enables a trusted entity, called *verifier*, to ask for an untrusted device, called *prover*, to provide assurance about its internal integrity. With its strong guarantees not suffering from false positives, remote attestation is becoming increasingly popular for critical embedded systems which can be used for medical, military or industrial control purposes. Previous proposals, which used checksums on static code regions to assure the load-time integrity, miss the runtime attacks that affect only dynamic memory regions. To address these attacks, this paper proposes a new scheme that attests the runtime integrity according to the control and data features of the program. The runtime check is performed in real time with the help of a novel hardware security module (HSM) which is connected to the prover's system bus. Proposed HSM detects runtime issues by checking compliance of the bits seen on the address and data bus with the static model loaded into its memory. Our attestation scheme can detect sophisticated runtime attacks such as code-reuse and non-control data attacks.**

***Index Terms*—remote attestation, runtime integrity, embedded systems security**

I. Introduction

Indispensable components of many critical infrastructures such as power grids, industrial control systems, health and transportation services, embedded devices pose significant risks because of their monitoring and control tasks which interact with the physical world in real-time. While their constrained nature—aiming specific tasks—hinders the deployment of most security solutions available to general-purpose high-end devices with operating systems (OS), the common use of unsafe languages (i.e., C/C++) for embedded software development inherits many vulnerabilities. Furthermore, the communication requirements via wired and wireless networks enable the adversaries to do physical harms remotely, which is a grave risk that threatens the states, companies, and individuals.

Remote attestation aims to address these risks by detecting compromised devices and reporting them to a remote entity. *Prover*, as a potentially infected device, has to assure a remote party called *verifier* that the device is in a legitimate state. In a typical attestation procedure, the verifier makes an attestation request to the prover with a challenge. Then, the prover performs some measurements on its memory contents and returns with a response generated based on the given challenge and the measurement.

In conventional attestation techniques, the prover provides a hash-based checksum of its memory contents to the verifier as proof of integrity. Although these measurements can ensure the integrity of static memory regions (i.e., code segments), their use on dynamic regions (e.g., heap, stack) is not practical since the verifier cannot feasibly know all legitimate states observable on these regions in advance. Therefore, static techniques fail to detect runtime attacks that only corrupt the data space via software exploits.

Early schemes such as DynIMA [1] and ReDAS [2] attest runtime behaviour of programs in high-end devices via measurement components relying on the OS. As these approaches are not adaptable to the embedded devices lacking an OS (e.g., bare-metal programming), putting trust on these complex systems (i.e., as part of the TCB) is another concern. More recent C-FLAT [3] and LO-FAT [4] schemes target embedded systems and propose to attest the control-flow of the programs. Both studies expect the verifier to discover all legitimate paths/hashes in advance which can be an issue for the programs with many control structures (i.e., path explosion). Moreover, despite the coverage of control-flow bending attacks, these schemes are not able to detect other variants of non-control data attacks due to the lack of data-flow attestation. A very recent paper LiteHAX [5] aims to address these drawbacks which is also the primary motivation of this paper. LiteHAX reports other non-control data variants via data-flow attestation. Despite not changing the architecture, LiteHAX proposes a pipeline integration with the CPU as a system-on-chip (SoC) design. It also requires a continuous online phase between the parties which increases the communication overhead.

While this paper addresses the shortcomings of C-FLAT and LO-FAT schemes such as coverage of data-only attacks and path explosion issues, differently from the LiteHAX, it does not require an online phase causing communication overhead, and it proposes a less invasive off-chip design via an external hardware module that can fit better into existing systems with legacy issues.

In line with this vision, this paper makes the following contributions:

1) Reporting of both control-flow hijack and non-control data attacks to a remote entity.

```
1: int authenticated=0;
2: char packet[1000];
3: while (!authenticated){
4:    PacketRead(packet);
5:    if (Authenticate(packet))
6:       authenticated=1;
7: }
8: if (authenticated)
9:    ProcessPacket(packet);
```

Figure 1: Vulnerable code to launch control-flow hijack and control-flow bending (non-control data) attacks

```
1:char cmd[][5]= {"ls", "pwd"};
2:char path[10]="/bin/";
3:char entry[2];
4:printf("1:%s\n2:%s\nPick", cmd[0],cmd[1]);
5:scanf("%s",  entry);
6:strcat(path,cmd[atoi(entry)-1]);
7:execl(path,cmd[atoi(entry)-1], (char*) NULL);
```

Figure 2: Vulnerable code to launch a data-only attack

2) External off-chip module design with a system bus integration addressing strong adversary assumptions and legacy issues of critical embedded systems.
3) Runtime monitoring logic that can perform in real-time with a resource-constrained hardware module.

II. Problem Setting

Threats to the runtime integrity can be categorised into two main classes as control-flow hijack and non-control data attacks. While the simplest type of the former, *code-injection* attacks use data space to load and execute the payload, due to the data execution prevention (DEP) mechanisms, these attacks have evolved into *code-reuse* attacks such as *return-oriented programming (ROP)* [6] where the payload is expressed via existing code segments. In general, control-flow integrity (CFI) is required to eliminate this class of attacks. Contrary to control-flow hijack attacks, the adversary can reach the desired goal via non-control data attacks (e.g., corruption of condition variables). As the *data-only* variants follow execution paths identical to legitimate scenarios, the ones producing infeasible traces are called *control-flow bending* attacks [7]. Despite the CFI compliance, systematic way of performing non-control data attacks can even achieve Turing-complete attacks (i.e., *data-oriented programming (DOP)* [8]), in case of a suitable vulnerability. To address all non-control data variants, potential solutions should check the integrity of program data or its flow. This paper proposes an attestation scheme capable of reporting both attack classes.

A. Runtime Attacks

To explain the runtime attacks addressed, we use source code level abstraction of vulnerabilities. The first vulnerable code (Figure 1) is a modified SSH vulnerability [9] used previously by Miguel et al. [10] and can form a basis for both control-flow hijack and control-flow bending variants of non-control data attacks. The second vulnerability (Figure 2) illustrates data-only variants of non-control data attacks.

The first vulnerability in Figure 1 is caused since `PacketRead` accepts unbounded input data and assigns it to `packet` array insecurely which allows the attacker to launch different attack classes with a goal of processing unauthenticated packets. Two possible code-reuse scenarios as control-flow hijack attacks can enable the attacker to reach the same purpose either replacing function's return address saved before the `PacketRead` call with the address of `ProcessPacket` function call or the instruction that sets the `authenticated` variable. As a non-control data attack, the adversary can reach the same goal by overwriting the adjacent stack addresses to the `packet` array to corrupt the value of `authenticated` variable. This is also called a control-flow bending attack which produces an infeasible path that cannot belong to a legitimate scenario (due to the skip of `authenticated=1`).

As a different subclass of non-control data attacks, a data-only attack can be performed by exploiting the buffer overflow bug of the vulnerable code in Figure 2 to overwrite other local variables in the stack. The adversary can use command picking stage (`scanf` function) to overwrite `cmd` commands which will be executed via `execl` function. This fabricated code gives us a non-control data attack scenario of which the path is indistinguishable from a legitimate run.

B. System Model

We consider two entities which are the *verifier* and the *prover*. As a trusted high-end device, the verifier does not have resource limitations and can perform expensive static analysis tasks on the source code or the binary. As a potentially infected party, the prover is a single-threaded embedded device with a low-cost hardware security module (HSM) connected to its system bus as seen in Figure 3. Since this study focuses on code-reuse and non-control data attacks, we assume that the prover ensures code immutability and DEP via separation of code (RX) and data (RW) segments (e.g., Harvard architecture). The program being attested is an embedded software/firmware of which the control-flow graph (CFG) can be modelled precisely without indirect control transfers. Its variables can be distinguished at runtime via corresponding memory instructions (i.e., no use of recursive functions).

The HSM (see Figure 4) has built-in hardware implementations of described runtime monitoring logic (Section IV) and necessary cryptographic components (i.e., a hash function and digital signature scheme with a key) which enable it to perform in real-time. HSM has its own memory resources. The first part contains a static model given by the verifier at the deployment phase—described as runtime integrity model (RIM) in detail in Section III—as a representation of the program. The second dynamic part is allocated to store the copies or digests of critical program variables (i.e., variable cells) and a shadow stack for return addresses. This part collaborates with the static

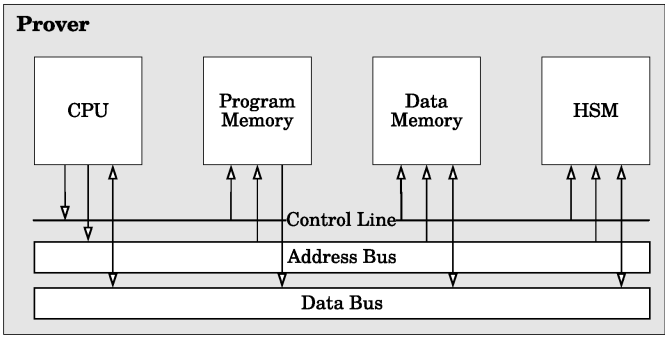

Figure 3: Prover's bus architecture of HSM design

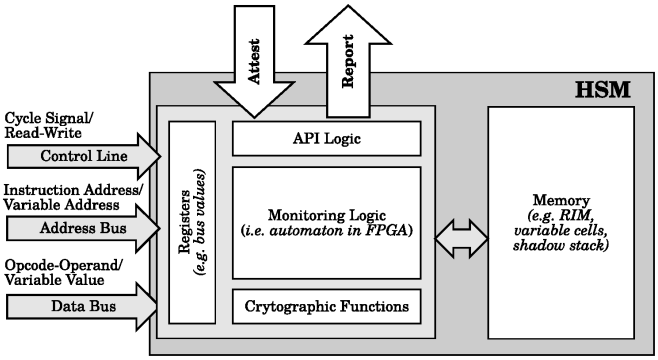

Figure 4: Overview of HSM design

model to check the integrity of selected program variables and control-flow. While the information captured through the bus represents behavioural input, the HSM provides a basic attestation API that reports the integrity violations of control and data features.

C. Adversary Model

Prior to the attestation, the adversary has access to the source code, binary and RIM. External resources are available to collect or record any protocol activity for later use. Only software attacks can be performed while physical attack capabilities are out of the scope. The adversary can exploit the vulnerable software to take control of the program (e.g., code-reuse attacks) or modify its data (i.e., non-control data attacks), though the adversary cannot affect the HSM's internal state and the verifier.

The ultimate goal of the adversary is malicious execution on the prover without being noticed by the verifier. Acting on the prover, the adversary can try to hide attack artefacts from the HSM. If this is not possible and the HSM has already detected an attack, the adversary may attempt to prevent the genuine reports from being received by the verifier and replace them with counterfeit but acceptable ones. The adversary can arbitrarily call the HSM's API to learn about the HSM's internal state or to generate signed attestation reports for later use. The adversary can intercept and modify any messages in the network or replay the responses sent earlier.

III. Design of Runtime Integrity Model (RIM)

For the program needs to be attested, the verifier generates a runtime integrity model (RIM) which corresponds to a static behavioural model of the program. This model is stored and used by the HSM as a guideline to check the integrity of control and data features of runtime information monitored on the system bus. As the model describes the legitimate control-flow paths for the program, it enables HSM to identify the memory instructions writing or reading on behalf of program variables by keeping track of the program flow. We define RIM as a directed graph $RIM = (S, T)$ consisting of the set S of nodes (*states*) and the set T of edges (*transitions*). Each node S_n corresponds to a block of instructions, and each edge T_n represents an instruction triggering a transition from one node to another one. Any program binary can thus be modelled as a directed graph. Since the HSM uses addresses to identify the instructions, a *state* (block) is a memory address range that does not contain any instruction worth explicit mentioning whereas a *transition* (edge) corresponds to a single instruction address that requires a change in the current state.

A. States and Transitions

To guide the HSM what action is required for each instruction (explained in Section IV), the RIM has different types of transitions and states. A transition as a single instruction either implies the change in the control flow or the presence/beginning of memory instructions for a program variable. States as instruction blocks sharing some common characteristics which are created to minimise the cost regarding both the model size and its execution. The RIM uses the following states and transitions types:

Non-data block (NB) states are sequences of instructions that do not change the control flow and do not contain a memory instruction of selected program variables.

Data block (DB) states represent the sequences of memory instructions of composite types (e.g., arrays). To be modelled as a data block, a composite variable should be used in the same way it is defined which means that the order and the number of elements should match for the corresponding write and read blocks.

Control (C, CR, CC) transitions are instructions that trigger a new state due to the change in the control flow. They either represent the call (CC) and return (CR) instructions or the target addresses of jump instructions (C).

Data (DP, DC) transitions describe the memory instructions of critical program variables. While primitive data transition (DP) corresponds to the standalone instruction of a primitive variable (e.g., integer), composite data transition (DC) represents the first instruction of a composite block such as an array which should be followed by a non-data block. Similar to the data blocks, both transition types contain annotated information specifying the access type (W:Write, R:Read) and the variable identifier (e.g., #1).

External function (EC) transition implies a function call (EC) to a dynamically linked library which is not explicitly represented by the RIM. Although the statically linked

```
0xxxx4eb <Authenticate>:
 ???????????dynmically linked library region???????????
0xxxx50e <PacketRead>:                                          S11
 xxxx50e:55              push  ebp                              S11
 .............9 instructions removed.....................      S11
 xxxx529:c3              ret                                    S11

0xxxx52a <ProcessPacket>:                                       S12
 xxxx52a:55              push  ebp                              S12
 .............9 instructions removed.....................      S12
 xxxx545:c3              ret                                    S12

0xxxx546 <main>:
 xxxx546:8d4c2404        lea   ecx,[esp+0x4]                    S0
 .............9 instructions removed.....................      S0
 xxxx565:c78508fcffff00  mov   DWORD PTR [ebp-0x3f8],0x0        S1
 xxxx56c:000000                                                 S1
 xxxx56f:eb32            jmp   xxxx5a3 <main+0x5d>              S1
 xxxx571:83ec0c          sub   esp,0xc                          S3
 .............2 instructions removed.....................      S3
 xxxx57b:e88effffff      call  xxxx50e <PacketRead>             S3
 xxxx580:83c410          add   esp,0x10                         S4
 .............3 instructions removed.....................      S4
 xxxx58d:e859ffffff      call  xxxx4eb <Authenticate>           S4
 xxxx592:83c410          add   esp,0x10                         S5
 xxxx595:85c0            test  eax,eax                          S5
 xxxx597:740a            je    xxxx5a3 <main+0x5d>              S5
 xxxx599:c78508fcffff01  mov   DWORD PTR [ebp-0x3f8],0x1        S6
 xxxx5a0:000000                                                 S6
 xxxx5a3:83bd08fcffff00  cmp   DWORD PTR [ebp-0x3f8],0x0        S2
 xxxx5aa:74c5            je    xxxx571 <main+0x2b>              S2
 xxxx5ac:83bd08fcffff00  cmp   DWORD PTR [ebp-0x3f8],0x0        S7
 xxxx5b3:7412            je    xxxx5c7 <main+0x81>              S7
 xxxx5b5:83ec0c          sub   esp,0xc                          S8
 .............2 instructions removed.....................      S8
 xxxx5bf:e866ffffff      call  xxxx52a <ProcessPacket>          S8
 xxxx5c4:83c410          add   esp,0x10                         S9
 xxxx5c7:b800000000      mov   eax,0x0                          S9
 .............13 instructions removed....................      S10
 xxxx5ef:c3              nop                                    S10
```

Figure 5: Disassembled output of the vulnerable program

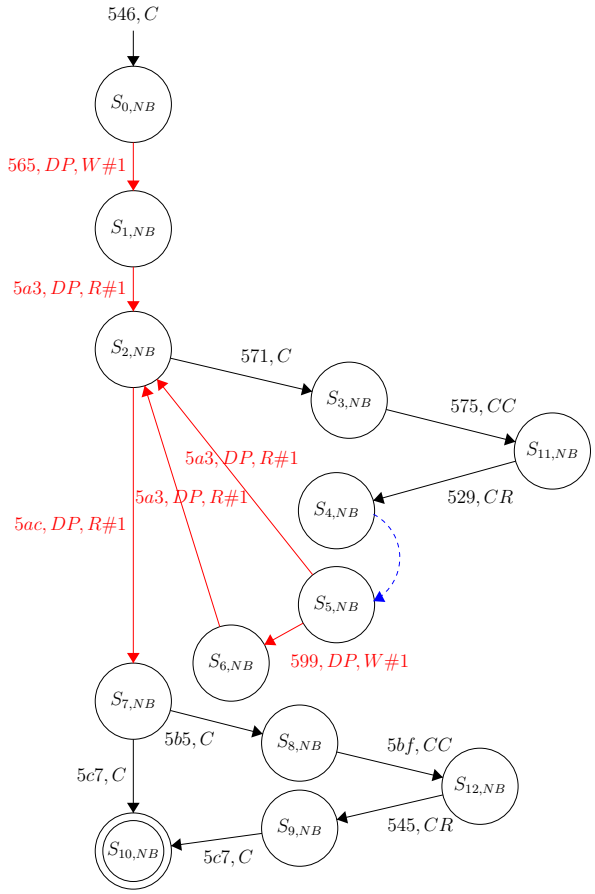

Figure 6: RIM of the vulnerable code

functions are handled as a monolithic code and modelled in detail, this transition type provides modularity for trustworthy dynamic libraries in return for limited guarantees about them. (i.e., lack of integrity checks on their non-control data).

B. Construction of the RIM

To extract the RIM, the verifier needs two types of information about the binary being attested. These are CFG of the program and the instructions using or defining program variables. Although binary analysis can approximate these, we assume that source code is available for a more precise modelling. By mapping the source code to machine instructions, the verifier extracts the instructions that operate for program variables or change the control flow while variable names are used to enumerate variable identifiers. Additionally, source code enables us to reduce the space required within the HSM for the RIM and the variable cells. As further source code analysis identifies critical variables for the program execution (i.e., variables only affecting the control flow and used by critical functions), liveness analysis allows the model to reuse HSM cells (identifiers) for different variables.

To demonstrate how the RIM represents the program binary, a concrete example of a binary (see Figure 5), is given in Figure 6. The model and the disassembled binary belong to the vulnerable code shown in Figure 1. The example illustrates the `authenticated` variable as a primitive data type, and `Authenticate` function as an external function of a dynamically linked library. As shown in Figure 6, while the first primitive data transition ($S_0 \rightarrow S_1$ via `565,DP,W#1`) represents the initialisation of the `authenticated`, the additional annotation on the edge ($W\#1$) specifies the type of operation (W:Write) and the identifier of the variable being written (#1 assigned for `authenticated`). Then, its use as the loop condition is represented by the transition (`5a3,DP,R#1`) and followed by S_2 where the control flow can diverge to two different paths afterwards. The following transitions correspond to either jump to the block of instructions inside the loop ($S_2 \rightarrow S_3$ via `571,C`) or the skip of the loop and execution of following if-condition ($S_2 \rightarrow S_7$ via `5ac,DP,R#1`). While the states S_{11}, S_{12} correspond to the `PacketRead`, `ProcessPacket` functions respectively, the blue edge as an external function call (`Authenticate`) represents a special mode of HSM's the monitoring logic which is explained in the following section. If the `Authenticate` function returns true, the instruction ($S_5 \rightarrow S_6$ via `599,PD,W#1`) sets the value of `authenticated` variable. Following transition chain ($S_7 \rightarrow S_8 \rightarrow S_{12} \rightarrow S_9 \rightarrow S_{10}$) illustrates the process of authenticated packet and the completion of the program, whereas $S_7 \rightarrow S_{10}$ path represents the completion of program

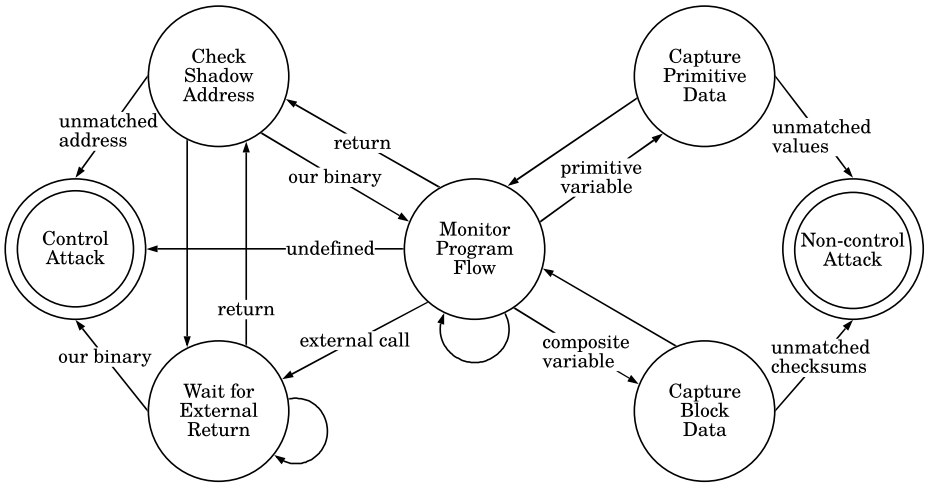

Figure 7: Bus-cycle based automaton of HSM modes

without processing the packet (though it is not a feasible path).

IV. Runtime Monitoring and Attack Detection

As the verifier constructs the RIM and loads it into HSM's memory, the system becomes ready to check the correctness of runtime behaviours observed through the system bus. While address bus provides the memory addresses of the instructions (executable) and the program variables (non-executable), bi-directional data bus carries the instruction itself (opcodes, operands) or the variable value being written to/read from memory (see Figure 4). The HSM uses address bus information to identify the binary instructions, whereas it requires the data bus to capture the values of program variables and to keep track of *return* and *call* instructions within the external function blocks of which we are ignorant. By using these as behavioural input and the RIM as a reference model, the HSM decides for the runtime integrity.

A. Runtime Integrity Check by the HSM

To attest the runtime, HSM's monitoring logic has five different modes where each completes its process in one bus cycle (see Figure 7). The HSM starts with *Monitor Program Flow* as the default mode which acts as a dispatcher. As this mode keeps track of program flow for data integrity checks, it attests control-flow with the help of *Check Shadow Address* and *Wait for External Return*. In case of an external call to a dynamically linked library, this mode changes to *Wait for External Return*. For an instruction annotated as variable read or write, it changes to either *Capture Primitive Data* or *Capture Block Data* to capture the variable values. If an integrity violation is detected while in any of these modes, the HSM stops further checking and waits to report the attack details. The verifier needs to perform a hard reset on the HSM to restart the process in a clean state. The Figure 8 illustrates the bus-cycle based logic of each mode of which the detailed explanations are given below:

Monitor Program Flow firstly identifies the segment of every address seen on the bus. For a non-executable address, this mode does not take any action and waits for the next bus cycle. For an executable one, it checks whether the address is within the range of current state (block) of the RIM or not. If so, depending on the type of current block, the HSM sustains the mode or switches to *Capture Block Data* mode in case of a data block. If the instruction address is not part of the block range, the HSM expects this address to be recognised by the outgoing edges (transitions) of RIM's current state where the return and call instructions additionally trigger shadow stack operations. In case of an unrecognised address, it sets the integrity violation flag for a control-flow hijack attack. Otherwise, the HSM updates the current state of the RIM as the transition dictates. Then, depending on the transition type, it picks the mode for the next bus cycle. For a jump, it maintains this mode. A primitive data transition triggers *Capture Primitive Data* whereas a composite data transition changes the mode to *Capture Block Data*. In case of an external function call, it switches to *Wait for External Return* mode where the current RIM state is updated and suspended until the expected block on the return.

Check Shadow Address checks whether the address to be returned on the data bus matches with the one popped from the shadow stack. As this mode reveals ROP attacks performed from our binary or dynamic libraries, it also brings the HSM back to normal *Monitor Program Flow* mode in case of a legitimate return to our binary.

Capture Primitive Data deals with memory instructions operating for primitive variables. To ensure integrity, the HSM uses reserved cells to store the exact copies of variable values. In case of a *write* instruction (access type is given by the RIM and the control line), the HSM captures the variable value on the data bus and records it to the reserved cell specified by the identifier information. After copying the value to HSM's memory, it switches back to the *Monitor Program Flow* mode. When this mode becomes active again due to *read* operation

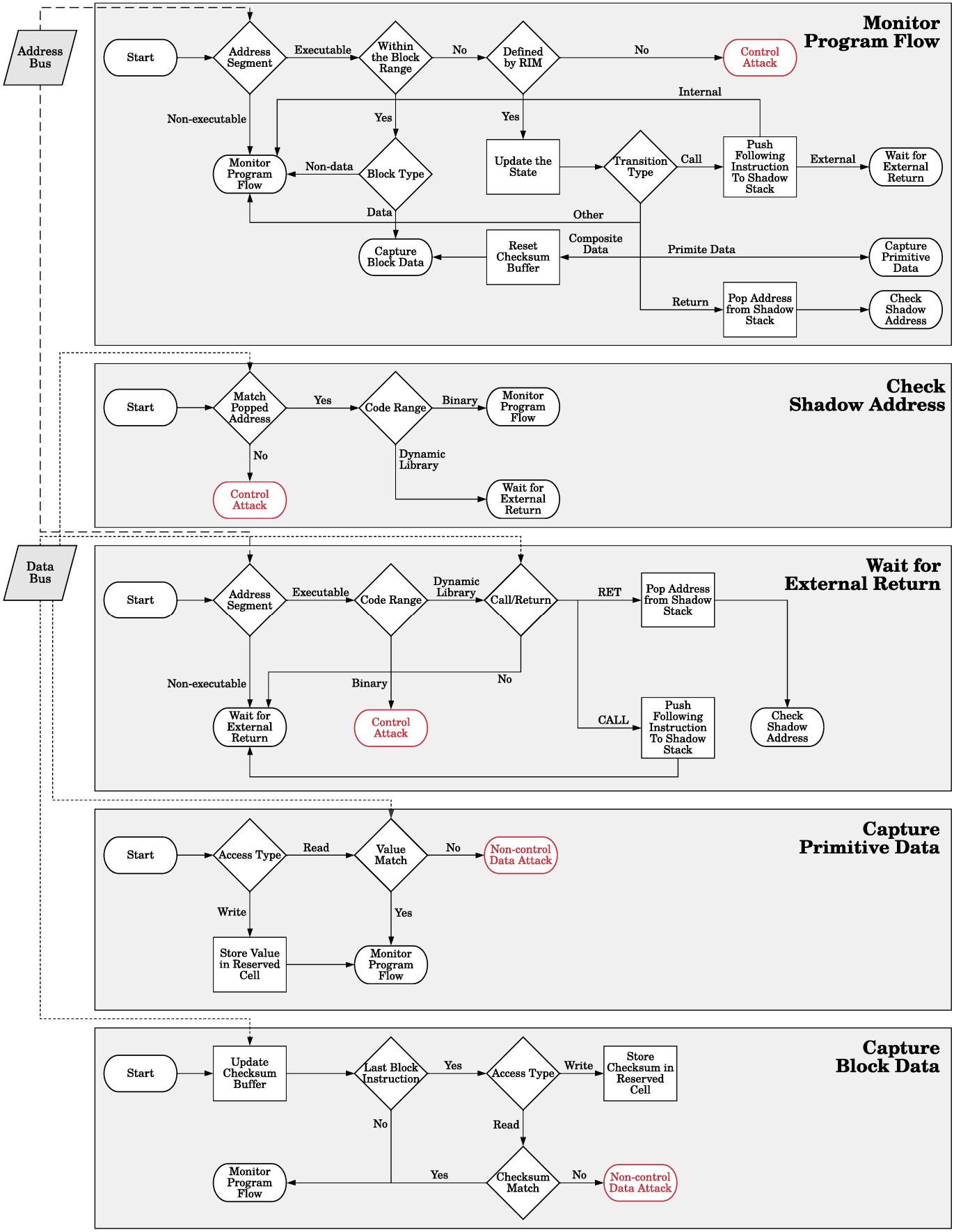

Figure 8: Detailed bus-cycle based process flow of HSM's monitoring logic using the RIM

of the same variable, the HSM compares the value seen on the data bus with the value stored previously. In case of a mismatch, it sets the flag for a non-control data attack.

Capture Block Data works similar to the *Primitive* mode. Differently, instead of storing separate copies of every element of a composite variable such as array type, it uses a checksum function (i.e., truncate of SHA-1) to digest the element values into a few HSM cells. When this mode is triggered by a transition for the first element of the variable, after resetting the checksum buffer, until the last instruction of the data block, the HSM continues to iterate the checksum on the buffer for each element value seen on the data bus via back-and-forths between this mode and *Monitor Program Flow* mode. As the last element is consumed, which is prior information given by the model, depending on the type of operation, the HSM either stores the final checksum value in the reserved cell for a writing block or checks the calculated checksum value with the previously recorded one for a reading block. If there is a mismatch, the HSM sets the flag for a non-control data attack.

Wait for External Return uses the address and data bus values together. For a non-executable address, it sustains this mode. For an executable one that belongs to our binary, it sets the control-flow attack flag since the program counter can only switch back to the binary via a legitimate return, which has to be monitored. For an address of external library, the HSM monitors the data bus for `call` or `ret` opcodes to check the integrity of possible nested call-return pairs with help of *Check Shadow Address* mode. In case of a call, it pushes the incremented address value (following instruction) to the shadow stack which is later popped for a return. In case of a return, it triggers *Check Shadow Address* to check whether the address on the next cycle matches with the popped one. Despite the reveal of ROP attacks, this mode does not promise to detect non-control data attacks that can occur within the dynamic library range which should be ideally trustworthy.

B. Attacks Coverage

Our scheme addresses both control-flow hijack and non-control data attacks. While the corruption of existing code segments and injection of malicious code to data segments already addressed by the features described in our system model, *Monitor Program Flow, Check Shadow Address* and *Wait for External Return* modes collaborate to detect code-reuse scenarios that reuse our binary range. In case of an external function call, the scheme does not promise to detect non-control data attacks that can be completed within the dynamic library's code range. For the programs where all the functions are statically linked and modelled by the RIM, *Capture Primitive/Block Data* modes collaborating with the *Monitor Program Flow* mode detect non-control data attacks including both data-only and control-flow bending variants.

Regarding non-control data attacks, we need to emphasise that even though the HSM does not notice the adversary at the time of corruption of a variable, it reveals any malicious attempt that relies on the use of the corrupted variables (which is equivalent to the definition of non-control data attacks [11]).

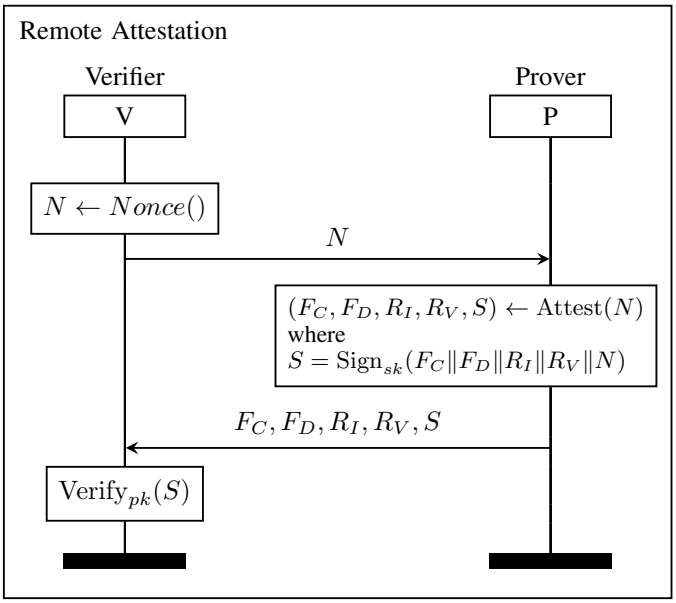

Figure 9: Overview of the attestation protocol

By linking the enumerated *def-use* instructions of program variables via their shadow values or digests within a legitimate control-flow context, HSM implicitly provides the same guarantees of data-flow integrity (DFI) [10] schemes—relying on *reaching definitions* lookups for variable uses—in a more efficient way. Since the HSM checks the value/digest integrity of variables, it does not promise to catch pure confidentiality issues (e.g., information leakage), unless the attack corrupts a pointer variable or violates the control-flow integrity. We need to stress that the value integrity means address integrity for pointer variables which can also refer to other value blocks of composite types. While this is not an issue for character pointers (i.e., strings) pointing blocks defined as read-only text, the HSM checks the integrity of other value blocks defined in the heap area (via malloc function) with the help of *Capture Block Data* mode, as long as the block elements are used in the same way that they are defined. Lastly, the variables defined and killed at CPU registers—as a result of compiler optimisations—are not represented by the RIM. As these optimisations reduces the cost of the RIM, they do not degrade the scheme's promises for non-control data attacks since the attacker will not be able to corrupt these variables via memory bugs that can only overwrite the data space.

V. Protocol Overview

This section presents a protocol design that assures the verifier receives a genuine report through an infected device and untrusted network. We consider that, at any moment, the verifier can make a request to learn about the prover's internal state. As seen in Figure 9, when the prover receives an attestation request containing a fresh nonce value N generated by the verifier, the prover calls the $Attest(N)$ provided by the HSM's API. Then, the prover needs to send back its output as the attestation response to the verifier. The response consists of

the control F_C and data F_D integrity violation flags, additional register information—to locate the vulnerability such as lastly executed instruction R_I and variable identifier used R_V before the violation—and the signature S of which all this information and the sent nonce are signed with a digital signature scheme. As the attestation response is received, the verifier firstly verifies the signature by using public-key of the HSM. While the key (sk) ensures the authenticity of the message, the signature content containing nonce N, flags and registers assure freshness and integrity properties of the message. Then, the verifier can check the sent flags to learn about the existence of an adversary and type or location of the attack. While the set F_C flag means a control-flow hijack attack violating CFI, F_D tells the existence of a non-control data attack that violates the DFI. Only if both flags are negative, the verifier can conclude there is not a runtime attack reported at the time of check.

VI. Security Analysis

For an adversary to successfully compromise the prover without being detected by the verifier, he must either hide the attack artefacts from the HSM or forge a valid attestation response when queried by the verifier. This section analyses both possibilities separately.

A. HSM Attacks

Due to the bus system integration, every instruction executed, and data transferred from/to memory will be accessible by the HSM. Because hardware attacks are out of scope, any software attack has to go through the system bus and will be checked by the HSM according to the RIM loaded by the verifier. Thus, for an attack to remain undetected by the HSM, the adversary either has to modify the prover's execution in a way that does not deviate from the RIM, or he needs to find a flaw in HSM's monitoring logic (Figure 8). The former contradicts the assumption that the RIM precisely captures the program behaviour consisting of control-flow and variable instructions.

Regarding the monitoring logic, if the adversary subverts the program's control flow, the *Monitor Program Flow* and *Check Shadow Address* modes are designed to identify it as the offending instruction is on the address bus. For the programs where all the functions are statically linked and modelled by the RIM, *Capture Primitive/Block Data* modes promise to detect non-control data attacks including both data-only and control-flow bending variants. While *Capture Primitive Data* mode detects any use of a corrupted primitive variable, composite variables (e.g., arrays) are protected by the created digests via *Capture Block Data* mode. The adversary can attempt to modify such a composite variable by finding a 2nd-preimage. The success probability of such an attack is dependent on the output length of the hash function. These modes are disabled only for the instructions of dynamically linked libraries (since they are not modelled by the RIM). However, the attacker cannot use this property to trick the HSM to execute instructions from our binary range without being detected. This is because *Wait for External Return* mode reports any instruction from our binary range without corresponding return instruction as an illegitimate control transfer from external code segments. Lastly, attempts to bypass the HSM via execution from data addresses or modification of the binary to comply with the RIM are not possible by assumption, due to the inherited code-immutability and data execution prevention features of the system model.

B. Protocol Attacks

The only option left to the adversary is to prevent the verifier from seeing the genuine violation flags. To accomplish this, the adversary has to return a valid response to the verifier's request. If the prover does not respond, the verifier will conclude the prover is compromised. Therefore, the adversary cannot simply block a message or remain silent after compromising the prover. There are only two ways an adversary can send a valid response: either replay a previously captured response or craft one from scratch. We look at each of these in turn.

The signature in the response (message 2, Figure 9) contains a nonce picked by the verifier. Thus, to replay the response message the adversary would either have to force the verifier to use the same nonce twice or predict what nonce is going to be used and query the prover ahead of time before compromising the prover to obtain a clean response. This is only possible with negligible probability since we do not allow the adversary to compromise the verifier and the nonce is chosen securely (i.e., uniformly from a large domain).

Thus, to return a valid message, the adversary must create it from scratch. However, the message is signed using a key stored in the HSM, which the adversary cannot obtain by assumption. Therefore, to forge the message the attacker has to break the existential unforgeability property of the underlying signature scheme which can be done only with negligible probability.

VII. Performance

To detect attacks in real-time, the scheme has to comply with the HSM's resource constraints and avoid unnecessary computations without degrading the guarantees. The HSM requires enough storage to host the RIM, the variable cells and the shadow stack. The size of the RIM can be defined as $\mathrm{O}(s+t)$ where s is the number of states and t is the number of transitions in the model. Both parameters are dependent on the number of instructions that change the control-flow or operate on behalf of the variables. We formulate them as:

$$\mathrm{s} = 2c + u + b, \quad \mathrm{t} = 2c + u + p + b$$

where c, u terms represent the number of conditional (e.g., `jne`,`je`) and unconditional control transfers (e.g, `call-ret`, `br`) respectively. p is the number of instructions that represents the use or definitions of primitive variables where b corresponds to the read/write blocks for composite variables. On the other hand, the space required for the variable cells can be described as the maximum number of live variables at any point of the program since the cell of a destroyed variable can be reused for a new variable. As a

	Binary					Variables		
	Count	Cond. (c)	Uncond. (u)	Read (r)	Write (w)	Global	Local	Functions
Total (-O0)	2763	7.8%	10.9%	12.7%	9.6%	14	74	39

Table I: Distribution of instruction types on the binaries and number of variables/functions found on the source code

worst-case scenario, this can be equal to the total number of program variables (i.e., in case all variables are defined as global). Since the recursive functions are out of scope, the size of the shadow stack is not an issue.

Although both the size of the RIM and the number of variables cells would be program specific, they can be approximated as a function of program size. To provide insight into such approximation, we have analysed an embedded code repository [12] providing the implementations of different system modules (UART, timer, ring buffer, etc.) for MSP430 launchpad. Table I summarises the ratio of control and memory instructions for the final image compiled without any optimisation and the number of variables corresponding to them. The metrics (i.e., $c = 7.8\%$, $u = 10.9\%$, $r = 12.7\%$ and $w = 9.6\%$) provide a feasible upper bound for the space complexity of the RIM as a linear function of binary size. Regarding the variable cells, the source analysis shows that the required number of cells is less than thirty for such firmware.

Regarding the monitoring logic, although each HSM mode has different process flows, their completions are still independent of the RIM's size with a constant time complexity $\mathrm{O}(1)$ since the RIM has two edges at most for conditional jumps and shadow stack is used for return addresses. We need to emphasise that each mode is designed to complete its task within the same bus cycle. For an instruction cycle consisting of one bus cycle (i.e., access to data space is not required), the HSM only executes the central *Monitor Program Flow* mode. In case of an instruction operating as two bus cycles (e.g., return, variable read/write), the second bus cycle is for *Check Shadow Address* or *Capture Primitive/Block Data* modes. To perform these checks in real-time, we consider a non-generic hardware-based implementation such as FPGA [13] for the monitoring logic described in Section IV and Figure 8. Implementation of the monitoring task—of which complexity is independent of the RIM—at a lower abstraction layer would enable the HSM to process much faster where the entire process of each bus mode can be completed in a single tick of the FPGA's clock.

VIII. Related Work

Early software-based attestation examples [14]–[16] rely on checksums calculations via pseudorandom memory traversals (initiated by the nonce) where the verifier expects an undelayed response which otherwise interpreted as the hiding attempt of the adversary. Although hardware-assisted [17]–[19] checksum calculations can address stronger adversaries with a hardware root of trust, similar to the software-based techniques, they are vulnerable to runtime attacks since checksum-based approaches can only attest the integrity of static memory regions. Trusted Execution Environments (TEE) such as Intel SGX [20] and ARM TrustZone [21] can provide isolation for security sensitive tasks as dynamic roots of trust, and can mitigate some runtime issues (i.e., hardened ROP attacks). However, the isolation does not eliminate the risks of a vulnerable application completely (e.g., a non-control data attack corrupting adjacent variable) as they do not fit the legacy devices.

DynIMA [1] extends the static IMA [18] by introducing a taint tracking system for untrusted data to alert when used as a code pointer. ReDAS [2] attests the runtime behaviour based on the fulfilment of two kinds of system properties which are formulated as *structural integrity* (e.g., any return address has to be the instruction address following its call instruction) and *global data integrity* constraints (e.g., data invariants). In addition to the concern of putting the complex operating systems in the TCB, both studies target high-end devices differently from the devices aimed here. C-FLAT [3] and LO-FAT [4] target embedded devices as the state-of-the-art runtime attestation schemes. Both studies use cumulative hash values to digest the execution traces for control-flow attestation reporting code-reuse and control-flow bending attacks. While C-FLAT records the actual control flow via binary instrumentation, LO-FAT proposes hardware extensions for branch monitoring. However, as both studies do not address data-only attacks, they require the verifier to discover all legitimate paths in advance which brings path explosion issue for the software with many control structures, though the LO-FAT eases the issue by separating loop metadata. A newer scheme, LiteHAX [5] detects other variants of non-control data attacks. LiteHAX records the control-flow trace as bitstream and creates hashes only for memory store/load instructions to attest the data-flow where the verifier later reproduces the same hash of store/load instructions with the help of bitstream record. Due to its online design, LiteHAX proposes more communication overhead differently from our scheme.

Instead of reporting to a remote entity, there are also prevention mechanisms for runtime attacks. Software-based CFI by Abadi et al. [22] introduces inlined enforcement for the destinations control transfers, whereas Kuznetsov et al. [23] propose less overhead via an isolated region for code pointers. Hardware-assisted solutions such as HCFI [24] and HAFIX [25] extend the instruction set (ISA) and require compiler modifications. Despite their benefits over software-based approaches such as stronger protection and less overhead, their on-chip design increases deployment costs for existing devices with legacy issues. For DFI, pioneering software-based solution [10] uses *reaching definitions* analysis to mitigate both control-hijack and non-control data attacks by maintaining a runtime definitions table (RDT). This table is used to record write accesses on memory addresses for a later

search during the read accesses to decide whether they comply with the static information extracted. Hardware-based solution relying on data-flow isolation, HDFI [26] proposes a less costly approach. Instead of checking instruction identifiers, HDFI splits memory addresses as sensitive and non-sensitive via one-bit tags to apply more coarse-grained policies in return of a precision loss. Since these solutions approximate memory safety better than CFI techniques, they can mitigate non-control data attacks. As our scheme reports control-flow hijack attacks, by linking the *def-use* instructions of program variables via their values or digests within a legitimate control-flow context checked already, it provides the same coverage of a DFI scheme for non-control data attacks instead of applying expensive access policies to each memory address.

IX. Conclusion

This paper presents a novel remote attestation scheme that reports the control and data integrity issues together. Without having to discover all execution paths in advance—bringing path explosion—or causing significant communication overhead between parties, the scheme detects code-reuse and non-control data attacks in real time by proposing an off-chip hardware module. While the scheme monitors the control flow in an automaton fashion, with the help of a few variable cells, the scheme detects non-control data issues under tight memory constraints. As its off-chip design with system bus integration makes it adaptable for critical embedded systems with legacy issues, it monitors runtime behaviour from a point that adversary cannot hide without a physical attack.

References

[1] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 49–54, 2009.

[2] C. Kil, E. C. Sezer, A. M. Azab, P. Ning, and X. Zhang, "Remote attestation to dynamic system properties: Towards providing complete system integrity evidence," *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 115–124, 2009.

[3] 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 - CCS'16*, 2016, pp. 743–754.

[4] G. Dessouky, S. Zeitouni, T. Nyman, A. Paverd, L. Davi, P. Koeberl, N. Asokan, and A.-R. Sadeghi, "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware," in *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 2017, p. 24.

[5] G. Dessouky, T. Abera, A. Ibrahim, and A.-R. Sadeghi, "Litehax: lightweight hardware-assisted attestation of program execution," in *Proceedings of the International Conference on Computer-Aided Design*. ACM, 2018, p. 106.

[6] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*. ACM, 2007, pp. 552–561.

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

[8] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks," in *Proceedings - 2016 IEEE Symposium on Security and Privacy*. IEEE, 2016, pp. 969–986.

[9] "SSH CRC-32 Compensation Attack Detector Vulnerability." [Online]. Available: http://www.securityfocus.com/bid/2347

[10] C. Miguel, M. Costa, and T. Harris, "Securing Software by Enforcing Data-flow Integrity," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 147–160.

[11] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *USENIX Security Symposium*, vol. 5, 2005.

[12] "Simply Embedded." [Online]. Available: https://github.com/simplyembedded/msp430_launchpad

[13] M. D. James, "A Reconfigurable Trusted Platform Module," Ph.D. dissertation, 2017. [Online]. Available: https://scholarsarchive.byu.edu/etd

[14] A. Seshadri, A. Perrig, L. Van Doom, and P. Khosla, "SWATT: SoftWare-based ATTestation for embedded devices," *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2004, pp. 272–282, 2004.

[15] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying Code Integrity and Enforcing Untampered Code Execution on Legacy Systems," in *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, 2005, pp. 1–16.

[16] T. AbuHmed, N. Nyamaa, and D. H. Nyang, "Software-based remote code attestation in wireless sensor network," in *GLOBECOM - IEEE Global Telecommunications Conference*, 2009.

[17] E. Brickell, J. Camenisch, and L. Chen, "Direct anonymous attestation," in *Proceedings of the 11th ACM conference on Computer and communications security - CCS '04*, 2004, p. 132.

[18] R. Sailer, X. Zhang, T. Jaeger, and L. V. Doorn, "Design and Implementation of a TCG-based Integrity Measurement Architecture," in *USENIX Security Symposium*, vol. 13, 2004, pp. 223–238.

[19] K. El Defrawy, A. Francillon, D. Perito, G. Tsudik, K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust." *Ndss*, 2012.

[20] Intel Corporation, "Intel® Trusted Execution Technology - Software Development Guide," 2017. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf

[21] ARM, "ARM Security Technology: Building a Secure System using TrustZone Technology," p. 108, 2009. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf

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

[23] V. Kuznetsov, L. Szekeres, and M. Payer, "Code-pointer integrity," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, no. October, 2014, pp. 147–163.

[24] N. Christoulakis, G. Christou, E. Athanasopoulos, and S. Ioannidis, "HCFI: Hardware-enforced Control-Flow Integrity," in *Codaspy*, 2016, pp. 38–49.

[25] L. Davi, M. Hanreich, D. Paul, A.-r. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, and Y. Jin, "HAFIX: Hardware-Assisted Flow Integrity eXtension," in *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*, 2015, pp. 1–6.

[26] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, "HDFI: Hardware-Assisted Data-Flow Isolation," in *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 2016, pp. 1–17.