OAT: Attesting Operation Integrity of Embedded Devices

Zhichuang Sun
Northeastern University

Bo Feng
Northeastern University

Long Lu
Northeastern University

Somesh Jha
University of Wisconsin

***Abstract*—Due to the wide adoption of IoT/CPS systems, embedded devices (IoT frontends) become increasingly connected and mission-critical, which in turn has attracted advanced attacks (e.g., control-flow hijacks and data-only attacks). Unfortunately, IoT backends (e.g., remote controllers or in-cloud services) are unable to detect if such attacks have happened while receiving data, service requests, or operation status from IoT devices (remotely deployed embedded devices). As a result, currently, IoT backends are forced to blindly trust the IoT devices that they interact with.**

To fill this void, we first formulate a new security property for embedded devices, called *"Operation Execution Integrity"* or *OEI*. We then design and build a system, OAT, that enables remote OEI attestation for ARM-based bare-metal embedded devices. Our formulation of OEI captures the integrity of both control flow and critical data involved in an operation execution. Therefore, satisfying OEI entails that an operation execution is free of unexpected control and data manipulations, which existing attestation methods cannot check. Our design of OAT strikes a balance between prover's constraints (embedded devices' limited computing power and storage) and verifier's requirements (complete verifiability and forensic assistance). OAT uses a new control-flow measurement scheme, which enables lightweight and space-efficient collection of measurements (97% space reduction from the trace-based approach). OAT performs the remote control-flow verification through abstract execution, which is fast and deterministic. OAT also features lightweight integrity checking for critical data (74% less instrumentation needed than previous work). Our security analysis shows that OAT allows remote verifiers or IoT backends to detect both control-flow hijacks and data-only attacks that affect the execution of operations on IoT devices. In our evaluation using real embedded programs, OAT incurs a runtime overhead of 2.7%.

I. INTRODUCTION

Internet-of-Things (IoT) and Cyber-Physical Systems (CPS) are being rapidly deployed in smart homes, automated factories, intelligent cities, and more. As a result, embedded devices, playing the central roles as sensors, actuators, or edge-computing nodes in IoT systems, are becoming attractive targets for cyber attacks. Unlike computers, attacks on embedded devices can cause not only software failures or data breaches but also physical damage. Moreover, a compromised device can trick or manipulate the IoT backend (e.g., remote controllers or in-cloud services): hijacking operations and forging data.

This work was supported by the National Science Foundation (Grant#: CNS-1748334), the Office of Naval Research (Grant#: N00014-17-1-2227), and the Army Research Office (Grant#: W911NF-17-1-0039).

Unfortunately, today's IoT backends cannot protect themselves from manipulations by compromised IoT devices. This is due to the lack of a technique for remotely verifying if an operation performed by an IoT device has been disrupted, or any critical data has been corrupted while being processed on the device. As a result, IoT backends are forced to blindly trust remote devices for faithfully performing assigned operations and providing genuine data. Our work aims to make this trust verifiable, and therefore, prevent compromised IoT devices from deceiving or manipulating the IoT backend.

We take the general approach of remote attestation, which allows a device to prove its integrity (with regard to certain security properties) to a remote verifier. Although a large body of works on remote attestation exists [54], [60], [61], [55], [12], [36], they have different goals from ours. Moreover, most of them are focused on verifying basic security properties, such as static code text integrity, and therefore cannot capture the advanced attacks that are becoming mainstream recently. For example, return-oriented programming (ROP) and data-only attacks are easy to launch on embedded devices, as demonstrated on vulnerable industrial robot controllers [51]. C-FLAT [2] took the first step towards control-flow integrity (CFI) attestation. But a major limitation of C-FLAT is the non-deterministic verifiability of its control-flow hashes (i.e., a given hash may not be verifiable due to the program path explosion issue). Moreover, C-FLAT does not check data integrity (i.e., data-only attacks are not covered).

In this paper, we introduce the first attestation method that captures both control-flow and data-only attacks on embedded devices. Using this method, IoT backends can now verify if a remote device is trustworthy when it claims it has performed an operation, sent in a service request, or transported back data from the field. In addition, unlike traditional attestation methods, which only output a binary result, our method allows verifiers to reconstruct attack execution traces for postmortem analysis.

Our attestation is based on a new security property that we formulated, called *Operation Execution Integrity* (OEI), which combines both the control-flow integrity and critical data integrity of an "*operation*" (i.e., a self-contained task or logic). An operation satisfies OEI if and only if the operation was performed without its control flow altered or its critical data corrupted during the execution. For an operation to be attested, the IoT device (i.e., prover) sends an unforgeable

OEI *measurement* to the IoT backend (i.e., verifier), along with any output from the operation. The backend then checks the measurement to determine if OEI was satisfied during the operation. The backend accepts the operation output from the device only if the check passes (i.e., the received data or request is trustworthy).

OEI takes advantage of the "operation-oriented" design of embedded programs: code is typically organized in logically independent operations, such as moving a robotic arm, injecting a dose of medicine, sensing temperature, etc. Rather than covering an entire program, OEI is focused on the execution of individual operations (hence the name). This per-operation property allows for on-demand and efficient attestation on embedded devices without sacrificing security (§III-B).

We design and implement OAT (OEI ATtester), a system that enables OEI attestation on ARM-based embedded devices. It consists of a customized compiler for building attestation-enabled binaries, a runtime measurement engine running on IoT devices, and a verification engine for IoT backends. OAT addresses two key challenges associated with OEI attestation, or any remote attestation of control-flow and data integrity:

(1) Incomplete verification of control-flow integrity: Conventional hash-based attestation [2] can only verify a (small) subset of program executions (i.e., incomplete verification of control-flow). It is because this approach checks a given control-flow hash against a limited set of hashes pre-computed from known-legitimate program runs. This static hash pre-computation can never cover all possibilities due to program path explosions, even for small programs. As a result, this attestation cannot verify control-flow hashes, legitimate or not, outside of the pre-computed set.

We design a hybrid control-flow attestation scheme for OAT, which combines hashes and compact execution traces. This scheme enables complete control-flow verification as well as attack flow reconstruction, at the cost of a mildly increased measurement size. Our attestation scheme is partly inspired by the tracing-based CFI enforcement [24], [59]. But unlike previous work, which requires hardware tracing modules unavailable on deployed or debugging-disabled embedded devices, our scheme uses its own software-based tracing technique. Moreover, thanks to the combined use of hash and traces, OAT's space overhead is only a tiny fraction (2.24%) of tracing-based CFI's overhead. In addition, OAT checks both forward- and backward-edges. We discuss the details in §IV.

(2) Heavy data integrity checking: The existing data integrity checkers [13], [3], [11] have to instrument every memory-write instruction and sometimes memory-read instructions in a program. The heavy and extensive instrumentation is needed because these checkers have to decide during runtime, for every instrumented instruction, whether the instruction is allowed to store/load data to/from the referenced address. We call this *address-based checking*, which is too heavy for embedded devices.

OAT uses a novel data integrity checking technique. First, it only covers critical variables because not all program data is relevant to an operation to be attested. Critical variables are those that may affect the outcome of an operation. They are automatically detected by OAT or annotated by developers. Second, instead of address-based checking, our technique performs *value-based checking*. It checks if the value of a critical variable at an instrumented load instruction (i.e., use) remains the same as the value recorded at the previous instrumented store instruction (i.e., define). It only instruments the instructions that are supposed to access the critical variables, rather than instrumenting all memory-accessing instructions as address-based checkers would, even when only selected variables need checking. Our technique on average requires 74% fewer instrumentation than address-based checking does. We call this "Value-based Define-Use Check", which is discussed in §V.

Using OAT, IoT backends can now for the first time remotely verify operations performed by IoT devices. Our security analysis (§VIII-C) shows that OAT detects both control and data manipulations that are undetectable by existing attestation methods for embedded devices. Our performance evaluation (§VIII), based on real embedded programs, shows that OAT, on average, slows down program execution by 2.73% and increases the binary size by 13%. In summary, our work makes the following contributions:

- We formulate a new security property, OEI, for IoT backends to attest the integrity of operations executed on remote IoT/embedded devices. It covers both control-flow and critical data integrity.
- We design a hybrid attestation scheme, which uses both hashes and execution traces to achieve complete control-flow verification while keeping the size of control-flow measurements acceptable for embedded devices.
- We present a light-weight variable integrity checking mechanism, which uses selective and value-based checking to keep the overhead low without sacrificing security.
- We design and build OAT to realize OEI attestation on both the prover- and verifier-side. OAT contains the compile-time, run-time, and verification-time components.
- We evaluate OAT on five real-world embedded programs that cover broad use scenarios of IoT devices, demonstrating the practicality of OAT in real-world cases.

II. Background

A. Attacks on IoT Devices and Backends

Embedded devices, essential for IoT, have been increasingly targeted by powerful attacks. For instance, hackers have managed to subvert different kinds of smart home gadgets, including connected lights [53], locks [31], etc. In industrial systems, robot controllers [51] and PLCs (Programmable Logic Controller) [22] were exploited to perform unintended or harmful operations. The same goes for connected cars [42], [34], drones [30], and medical devices [23], [52]. In addition, large-scale IoT deployments were compromised to form botnets via password cracking [56], and recently, vulnerability exploits [40].

Meanwhile, advanced attacks quickly emerged. Return-oriented Programming (ROP) was demonstrated to be realistic on RISC [8], and particularly ARM [39], which is the common architecture for today's embedded devices. Data-only attacks [15], [33] are not just applicable but well-suited for embedded devices [62], due to the data-intensive or data-driven nature of IoT.

Due to the poor security of today's embedded devices, IoT backends (e.g., remote IoT controllers and in-cloud services) are recommended to operate under the assumption that IoT devices in the field can be compromised and should not be fully trusted [57]. However, in reality, IoT backends are often helpless when deciding whether or to what extent it should trust an IoT device. They may resort to the existing remote attestation techniques, but these techniques are only effective at detecting the basic attacks (e.g., device or code modification) while leaving advanced attacks undetected (e.g., ROP, data-only attacks, etc.). As a result, IoT backends have no choice but to trust IoT devices and assume they would faithfully execute commands and generate genuine data or requests. This blind and unwarranted trust can subject IoT backends to deceptions and manipulations. For example, a compromised robotic arm can drop a command yet still report a success back to its controller; a compromised industrial syringe can perform an unauthorized chemical injection, or change an authorized injection volume, without the controller's knowledge.

Our work enables IoT backends to reliably verify if an operation performed by a device has suffered from control or data attacks. It solves an important open problem that IoT backends currently have no means to determine if data, results, or requests sent from (insecure) IoT devices are trustworthy. Moreover, it allows backends to reconstruct attack control flows, which are valuable for forensic analysis.

B. ARM TrustZone

Our system relies on ARM TrustZone to establish the TCB (Trusted Computing Base). TrustZone is a hardware feature available on both Cortex-A processors (for mobile and high-end IoT devices) and Cortex-M processors (for low-cost embedded systems). TrustZone renders a so-called "Secure World", an isolated environment with tagged caches, banked registers, and private memory for securely executing a stack of trusted software, including a tiny OS and trusted applications (TA). In parallel runs the so-called "Normal World", which contains the regular/untrusted software stack. Code in the Normal World, called client applications (CA), can invoke TAs in the Secure World. A typical use of TrustZone involves a CA requesting a sensitive service from a corresponding TA, such as signing or securely storing a piece of data. In addition to executing critical code, TrustZone can also be used for mediating peripheral access from the Normal World.

OAT measurement engine runs as a TA in the Secure World. Its code and data, including collected measurements, are naturally protected. TrustZone allows provisions of per-device private keys and certificates, which enable straightforward authentication and signed/encrypted communication between a measurement engine (i.e., prover) and a remote verifier. During an active attestation phase, the instrumented code in the Normal World reports raw measurements to the Secure World, where the raw measurements are processed and signed. The final report (aka. measurement blob) along with the signature is handed back to the Normal World agent, which sends the report to the remote verifier. On our target platforms (i.e., ARM-based bare-metal embedded devices), TrustZone is the only feasible TCB option that can support basic remote attestation operations. Our evaluation (§VIII) shows that the end-to-end overhead of our system is acceptable, thanks to our efficient attestation scheme.

```
1  // Simplified control loop for robotic arms
2  void main_looper() {
3    // Command from remote controller
4    cmd_t cmd;
5    // Pointer to operation function
6    int (*op_func)(int, char*);
7    // Input from peripheral sensor
8    char peripheral_input[128];
9    int st = 0;
10   while(1) {
11     if (read_command(&cmd)) {
12       st = get_input(peripheral_input); //BUGGY!
13       if (status_OK(st)) {
14         // perform the operation
15         op_func = get_op_func(cmd->op);
16         (*op_func)(cmd->p_size, cmd->param);
17       }
18     }
19     usleep(LOOPER_SLEEP_TIMER);
20   }
21   return;
22 }
23 ...
24 // The operation that moves the robotic arm
25 int op_move_arm (int, char*) {...}
26 ...
```

Listing 1: An example of a control loop in a robotic arm

III. Design Overview

A. Example: A Vulnerable Robotic Arm

Before we discuss OEI and the attestation, we first present a simple example to demonstrate the problem. The vulnerabilities shown in this example were drawn from real embedded programs. This example is also referenced in a later discussion on how OEI attestation can be easily applied to existing code.

In this example of a vulnerable robotic arm (Listing 1), the function `main_looper` first retrieves an operation command from a remote controller and stores it in `cmd` (Line 11). The looper then reads a peripheral sensor (Line 12), which indicates if the arm is ready to perform a new operation. If ready, the looper finds the operation function specified by `cmd->op` (Line 15) and calls the function with the parameters supplied in `cmd->param` (Line 16). One such function (Line 25) moves the arm to a given position.

We introduce an attacker whose goal is to manipulate the operation of the robotic arm without being detected by the remote controller who uses the existing attestation methods. For simplicity, we assume the attacker has tampered with the sensor and uses it to feed exploit input to the robotic arm. This is realistic given that such external peripherals are difficult to authenticate and protect. The target of the attacker is Function `get_input` (Line 12), which contains a buffer overrun. The vulnerability allows malformatted input to be copied beyond the destination buffer (`peripheral_input`) into the subsequent stack variables (e.g., `cmd`).

By crafting input via the compromised sensor, the attacker can launch either control hijacking or data-only attacks on the robotic arm. To hijack the control, the attacker overwrites both `cmd->param` and `cmd->op` as a result of the buffer overrun exploit, which leads to the execution of an arbitrary operation. To mount a data-only attack, the attacker only needs to change `cmd->param` while leaving `cmd->op` intact, which turns the authorized operation harmful.

Though seemingly simple, such control and data manipulations on embedded devices are realistic and can cause severe consequences in the physical world. More importantly, existing remote attestation methods cannot detect such attacks because most of them are focused on static code integrity and none addresses dynamic data integrity. Therefore, the remote controller is unable to find that the robotic arm is compromised and did not perform the operation as commanded. Moreover, after receiving an (unverifiable) operation-success message from the compromised robotic arm, the controller may command other robotic arms to perform follow-up operations, which can cause further damage. This example illustrates the need for a new form of attestation that allows IoT backends to remotely check the integrity of operations performed by IoT devices.

B. *Operation Execution Integrity (OEI)*

We propose "Operation Execution Integrity" (or OEI) as a security property for embedded devices. By verifying OEI, a remote verifier can quickly check if an *execution of an operation* suffered from **control-flow hijack** or experienced **critical data corruption**. We formulate OEI with two goals in mind: (*i*) enabling remote detection of aforementioned attacks; (*ii*) demonstrating the feasibility of an operation-oriented attestation that can detect to both control and critical data manipulations on embedded devices. Next, we formally define OEI and provide its rationale.

To avoid ambiguity, we informally define an **operation** to be a logically independent task performed by an embedded device. To declare an operation for attestation, programmers need to identify the entry and exit points of the operation in their code. For simplicity of discussion, we assume that every operation is implemented in a way that it has a single pair of entry and exit, $\langle Op_{entry}, Op_{exit}\rangle$ where Op_{entry} dominates Op_{exit} and Op_{exit} post-dominates Op_{entry} in the control flow graph. We do not pose any restriction on what code can an operation include or invoke, except that an operation cannot contain another operation.

Let $P = \{Op_1, Op_2, ...Op_n\}$ be an embedded program, composed of n operations, denoted as Op_i. $CFG(Op_i)$ is the statically constructed CFG (control flow graph) for Op_i (i.e., the CFG's root is Op_i's entry point). Let CV be the set of critical variables in P (i.e., variables affecting the execution of the operations). $D(v)$ and $U(v)$ are the def- and use-sites of a critical variable v: a set of statements where v is *defined* or *used*. $V_b(v, s)$ and $V_a(v, s)$ are the values of variable v immediately *before* and *after* statement s executes.

Definition 1. OEI: for an operation Op_i, its execution satisfies OEI $\iff$ ① the control flow complies with $CFG(Op_i)$ and maintains backward-edge integrity, and ② during the execution, $\forall cv \in CV$, the value of cv reaching each use-site is the same as the value of cv leaving the previous def-site, written as $V_b(cv, u) = V_a(cv, d)$, where $d \in D(cv)$ is the last define of cv prior to the use of cv at $u \in U(cv)$. △

OEI entails that the control-flows and critical data involved in an operation must not be tampered with.

Operation-scoped CFI (§IV): OEI's control-flow requirement (① in Def. 1) is not a simple adoption of CFI to embedded devices, which would incur impractical time and space overhead on those constrained devices. Our operation-scoped CFI takes advantage of the operation-oriented design of embedded programs. It applies to executions of individual operations, which represent a much smaller attestation scope than a whole program and allow for on-demand attestation. It also implies backward-edge integrity (i.e., returns not hijacked).

Critical Variable Integrity (§V): We call the second requirement of OEI (② in Def. 1) Critical Variable Integrity, or CVI. It dictates that the *values* of critical variables must obey *define-use consistency*. Compared with other data integrity checkers [13], [3], [11] CVI is different in two ways. First, CVI only concerns critical variables, rather than all program data. Second, CVI uses value-based checking, instead of address-based checking, to significantly reduce code instrumentation and runtime overhead. CVI applies to the entire program execution and is not scoped by attested operations. We provide a method to assist developers to automatically identify critical variables. We define critical variables and explain our value-based check in §V.

Secure & Optimized Combination: OEI combines CVI and operation-scoped CFI. These two sub-properties mutually complementary. Without CVI, CFI alone cannot detect data-only attacks or control-flow hijacks that do not violate CFG (e.g., [10]). Without CFI, CVI can be bypassed (e.g., by ROP). On the other hand, this combination yields better performance than independently performing control-flow and data-flow checks. This optimized combination allows for the detection of both control-flow and data-only attacks without enforcing full CFI and DFI. It is suited for embedded devices.

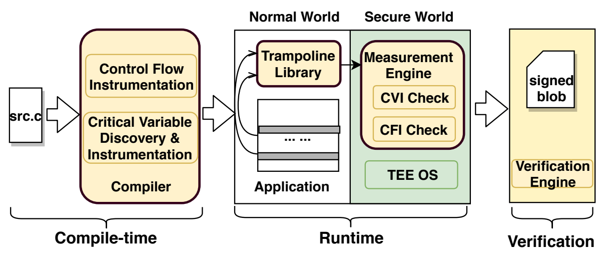


Fig. 1: The Workflow of OAT, whose components (colored in yellow) include the compiler, the trampoline library, the measurement engine in the Secure World (shown in green), and the remote verification engine.

Operation Verifiability: OEI caters to IoT's unique security needs. One defining characteristic of IoT devices is their frequent inter-operations with peers or cloud services. When a device finishes an operation, the operation initiator may wish to verify if the operation was executed as expected without interference. For example, a remote controller needs to verify if a robotic arm has performed a movement as instructed without experiencing any control or data manipulations. OEI attestation answers to such security needs of IoT, which are currently not addressed.

C. OAT Overview

We build OAT, a system to realize OEI ATtestation on ARM-based bare-metal embedded devices (i.e., no standalone OS in the Normal World). OAT consists of: (*i*) a customized **compiler** built on LLVM; (*ii*) a **trampoline** library linked to attestation-enabled programs; (*iii*) a runtime **measurement engine**; (*iv*) a remote/offline **verification engine** (Figure 1). For a given embedded program, the compiler identifies the critical variables and instruments the code for measurement collection. At runtime, the measurement engine processes the instrumented control-flow and data events and produces a proof or measurement, which the remote verifier checks and determines if the operation execution meets OEI. OAT relies on ARM TrustZone to provide the trusted execution environment for the measurement engine.

We design a hybrid attestation scheme that overcomes two challenges associated with CFI and data integrity verifications. First, remotely attesting CFI is more challenging than performing local CFI enforcement because remote verifiers can only rely on limited after-fact measurements to make the determination whereas local enforcers simply use the readily available unlimited runtime information. Furthermore, remote verifiers cannot always determine whether a hash-based control-flow measurement is legitimate or not because the verifiability is limited to those hashes pre-computed from known/traversed code paths.

Second, checking the integrity of dynamic data is known for its high overhead. Existing checkers [13], [3], [11], mostly address-based, instrument every memory-accessing instruction in a program, and during runtime, check if an instrumented instruction should be allowed to access the referenced address, based on a statically constructed access table. Even when the integrity checking is only applied to selected variables, address-based checkers would still need to instrument and check all memory-accessing instructions to ensure no unintended instructions can write to the addresses of the critical variables.

Our hybrid attestation scheme achieves complete verifiability while maintaining acceptable performance on embedded devices. For CFI attestation, OAT's measurements contain compact control-flow transfer traces for forward-edges and fixed-length hashes for backward-edges. This combination allows remote verifiers to quickly and deterministically reconstruct control flows. It also yields size-optimized measurements. For CVI, OAT performs local verification rather than remote attestation. By doing so, OAT avoids sending a large amount of data (e.g., critical values at def- and use-sites) to remote verifiers. Sending such data would be costly for IoT devices and undesirable when privacy is concerned.

To use OEI attestation, programmers declare the to-be-attested operations in their code by using two intuitive compiler directives: `#oei_attest_begin` and `#oei_attest_end`. They may also annotate critical variables of their choice via a GCC-style attribute. For example, to enable OEI attestation in Listing 1, a programmer only needs to change Line 4, 8, and 25:

```
4  cmd_t __attribute__((annotate("critical"))) cmd;
```

```
8  int __attribute__((annotate("critical")))
       peripheral_input = 0;
```

```
25 int op_move_arm (int, char*) {#oei_attest_begin
       ... #oei_attest_end}
```

For simplicity, our current design requires that a pair of `#oei_attest_begin` and `#oei_attest_end` is used in the same function (i.e., an operation enters and exits in the same call stack frame) and the `#oei_attest_end` always dominates the `#oei_attest_begin` (i.e., an operation always exits). Operations cannot be nested. These requirements are checked by our compiler. Developers are advised to keep the scope of an operation minimal and the logic self-contained, which is not difficult because most embedded programs are already written in an operation-oriented fashion.

As shown in Figure 1, during compilation, the customized compiler instruments a set of control flow transfers inside each to-be-attested operation. The compiler automatically annotates control-dependent variables as critical (e.g., condition variables). It also performs a global data dependency analysis on both automatically and manually annotated critical variables so that their dependencies are also treated as critical and subject to CVI checks. At runtime, the instrumented control flow transfers and critical variable define/use events trigger the corresponding trampolines, which pass the information needed for CFI or CVI verification to the measurement engine in the Secure World protected by TrustZone. Finally, the signed attestation blob (i.e., measurements) is sent to the remote verification engine (e.g., IoT backend) along with the output from the attested operation. We discuss OAT design details in §IV and §V.

D. Threat Model

We trust code running inside the Secure World (e.g., the measurement engine) and assume that attackers cannot break the TrustZone protection. We also trust our compiler and the trampoline code. We assume that attackers *cannot inject code in the Normal World or tamper with the instrumented code or the trampoline library*. This can be enforced using code integrity protection methods for embedded devices [16], [37], which are orthogonal to the focus of this paper, namely OEI attestation. Due to the absence of a standalone OS on bare-metal embedded devices, all code in the Normal World runs at the same privilege level (i.e., no higher privileged code exists or needs to be trusted).

On the other hand, we anticipate the following attacks in the Normal World, which previous attestation methods cannot detect. First, attackers may exploit vulnerabilities to launch ROP (return-oriented programming) and DOP (data-oriented programming) attacks. As a result, the control flow and the critical data of an embedded program can be compromised. Second, attackers may abuse unprotected interfaces of an embedded program and force the device to perform unintended or unauthorized operations. Our system is designed to detect these attacks by means of attestation. We present a security analysis on our system in §VIII-D.

IV. Operation-scoped Control-flow Attestation

Inspired by the operation-oriented nature of embedded programs, we attest CFI at the operation level, which avoids always-on measurement collection and whole-program instrumentation. It is lightweight and suitable for embedded devices.

Moreover, our attestation provides deterministic verifiability. It avoids the problem of unverifiable control-flow hashes, caused by code path explosions, as purely hash-based attestation faces [2]. The deterministic verifiability is achieved via a new hybrid measurement scheme, which uses a compact trace for recording forward edges and a hash for backward edges. This scheme resembles the hardware tracing-based CFI [24], [59]. But it has two major distinctions.

First, previous tracing-based CFI requires hardware components that are not available on deployed IoT or embedded devices[1]. Our control flow traces are generated purely using the software. Second, our trace is much shorter and more compact partly because it only records forward edges (i.e., backward edges or returns happen very frequently and thus would lead to overly long traces that embedded devices cannot store). By hashing the backward edges, rather than recording them in the trace, our scheme reduces the trace size by 97% (§VIII-B). Furthermore, combining trace and hash makes verification deterministic and free of path explosions. Verifiers no longer need to pre-compute or search for all possible code paths; they simply follow the forward-edge trace to reconstruct the actual execution path, and in the end, check the resulting backward-edge hash (more details later).

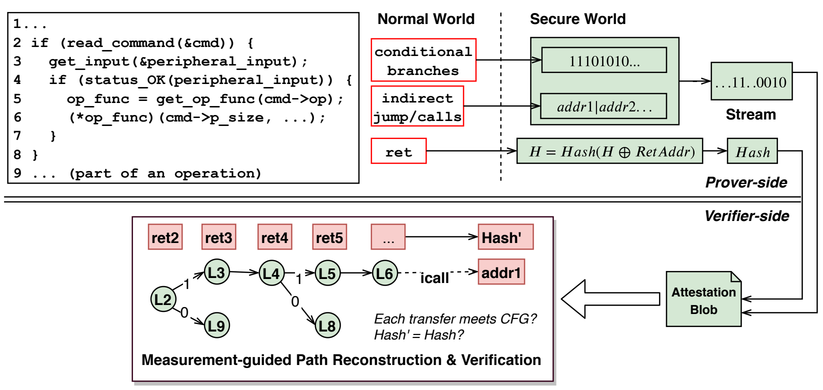
Instrumented Control Flow Events: OAT compiler instruments the code in each attestation-enabled operation to collect runtime measurements. We limit this instrumentation to a minimum set of the control flow transfers that need to be recorded/encoded in the measurement for deterministic verification. This minimum set, S_{min}, is constructed as follows. Consider all three types of transfers possible on a CFG: direct call/jump, conditional branch, and indirect transfers (indirect call/jump and return). Only the last two types need to be measured and are included in S_{min}. This is because knowing where each branch and indirect transfer went during a code execution is both sufficient and necessary for statically finding the exact operation execution path on the CFG. Direct calls and jumps are not included in S_{min} (i.e., no need for instrumentation) because their destinations are unique and statically determinate.

The instrumentation code simply calls the trampoline function corresponding to the type of the instrumented control-flow transfer, reporting the destination of the transfer. For a branch, its destination is one bit: *taken* (1) or *non-taken* (0). For an indirect transfer, its destination is the memory address. The trampoline functions are thin wrappers of the world-switching routine. When called, they initiate the switch to the Secure World and pass the destination information to the measurement engine.

Measurement Scheme: The measurement engine maintains two types of measurements for control-flow attestation: a **trace** and a **hash**. The trace is used for recording forward-edge control flow transfers (branches and indirect calls/jumps). The hash is for encoding backward-edge transfers (returns). The measurement engine updates the measurement trace or hash respectively, as shown in the upper half of Fig. 2. For each branch, it adds the taken/non-taken bit to the trace. For each indirect call/jump, it adds the destination address to the trace. Note that code addresses do not change across different firmware runs (including verification runs) because embedded firmware is always loaded at the pre-specified base address in memory. When dynamic loading or ASLR becomes available in embedded firmware, the measurement engine will need to record the firmware base address in the attestation blob, in order for the verifier to construct the same code layout in memory and check the trace. For each return, the measurement engine encodes the return address to the hash: $H = Hash(H \oplus RetAddr)$. Here we use the symbol $\oplus$ to represent a binary operation. In our implementation, we use concatenation[2]. The trace and the hash together form the

[1]Although recent MCUs support tracing, this optional feature is meant for debugging and usually unavailable on for-release devices due to additional hardware cost.

[2]See our formal definition and proof of the verification scheme in Appendix §B



```
1...
2 if (read_command(&cmd)) {
3   get_input(&peripheral_input);
4   if (status_OK(peripheral_input)) {
5     op_func = get_op_func(cmd->op);
6     (*op_func)(cmd->p_size, ...);
7   }
8 }
9 ... (part of an operation)
```

Fig. 2: Operation-scoped control-flow attestation. By measuring the control-flow of an executing operation, the measurement engine produces a control-flow proof that consists of a *trace* (proving forward edges including branches and indirect transfers) and a *hash* (proving backward edges or returns). This measurement scheme allows a remote verifier to statically and deterministically reconstruct the code path of the operation execution and perform full CFI verification.

attestation blob, which serves as the proof of the control flow of an executed operation.

In our design, we chose the cryptographic hash function BLAKE-2s[3] as the $Hash$ function for its high speed, security, and simplicity. BLAKE-2s has a block size of 64 Bytes and an output size of 32 Bytes. We present the collision analysis as part of the security analysis in § VIII-D.

The reason why we use two forms of measurements, namely trace and hash, is two-fold. First, a forward-edge trace allows for reconstruction and easy verification of recorded control-flow transfers. Second, a hash has fixed length and does not grow whereas a trace grows as the execution proceeds. However, control-flow hashes by themselves are not always verifiable due to the impossibility of pre-computing all possible code paths for a program and their hashes. Our measurement scheme uses the trace and hash in tandem to combine their strengths while avoiding their individual weaknesses, in terms of the ease of verification and space efficiency.

Recording the forward-edge trace is necessary for code path reconstruction. These events are either very compact (1 bit for each branch) or infrequent (indirect calls/jumps are less common than direct calls/jumps). Therefore, they do not bloat the trace. On the other hand, we encode return addresses in the hash because they are not needed during the path reconstruction phase (i.e., only needed for checking backward-edge CFI after a path has been constructed). Plus, returns happen fairly frequently and, if recorded in the trace, would consume too much space.

A possible (yet to implement) optimization would be to record the event type information in a separate sequence and then use this information to enable early detection of control flow divergences during verification (i.e., when a type mismatch is detected, the verifier can conclude that the current stream is invalid and terminate the verification early). For three types of events (*i.e.,* conditional branch, indirect branch and return), two bits would be enough for identifying and recording each type. This optimization can speed up the verification process at the cost of using extra storage. However, given the fact that RAM and flash storage on embedded devices are fairly limited and the verification process does not affect the runtime performance, we decided not to implement the early termination optimization for the current prototype.

[3]https://blake2.net/

The measurements are stored in a buffer allocated in the Secure World. Although rare, this buffer can run out if an operation execution is very long and yields a measurement trace longer than the buffer. When this happens, the measurement engine signs the current measurements, temporarily stores it in the flash storage, and frees up the buffer for subsequent measurements. At the end of the operation, the measurement engine sends all measurements to the remote verifier.

Measurement Verification: Given a control-flow proof for an operation execution (i.e., a trace and a hash), generated and signed by the measurement engine, the verification engine statically reconstructs the code path on the CFG of the operation, as shown in the lower half of Fig. 2. Starting from the root basic block, or the entry point, the verifier abstractly executes the operation on the CFG by following the forward-edge trace. During the abstract execution, the verifier maintains a simulated call stack to keep track of the return addresses and computes/updates the hash in the same way as the measurement engine does during runtime.

Specifically, when the abstract execution encounters a control-flow diverging point (i.e., more than one edge on the CFG can be followed) the verifier takes the next available element out of the trace and consults it for direction: either a taken/non-taken bit for a branch or an address for an indirect call/jump. The verifier also performs a CFI check in case of an address. A control flow violation is found when: ① the CFI check fails; ② a mismatch is observed between a basic block and the corresponding trace element (e.g., the current basic block ends with an indirect call but the next available element indicates a branch); or ③ after the abstract execution, the verifier-computed hash does not match the reported hash.

All indirect call/jump violating CFI trigger ①. ② can happen because when ROP happened during the actual execution, the trace did not return to the call site whereas the abstract execution always returns correctly (cued by the simulated call stack). This mismatch leads to early detection of ROP (i.e., no need to wait till ③). ③ signals that one or more return addresses (or backward control flow transfers) were corrupted during runtime. Note that not all ROP trigger ② but all ROP trigger ③.

V. Critical Variable Integrity

Data-only attacks, including data-oriented programming, are capable of manipulating program behavior and internal logic without injecting code or violating CFI [15], [33]. For example, as shown in Listing 1, a simple buffer overflow can make the robotic arm perform attacker-specified operations without breaking CFI. Unfortunately, existing attestation methods cannot detect data-only attacks.

We formulate Critical Variable Integrity (CVI) as a sub-property of OEI to detect data-only attacks on embedded devices. CVI is selective and concerns only data that attacks target: (*i*) **control-dependent data**, such as conditional variables, whose value directly determines the code path of a particular program execution; (*ii*) **semantically critical data**, whose value, though not directly affecting code execution paths, influences the outcome of an operation, such as `cmd` in Listing 1. CVI is not scoped by attested operations due to external data dependencies.

CVI is different from previous works on data integrity check [13], [3], [11], which require heavy instrumentation and is unsuitable for embedded devices. For example, DFI [13] uses a whitelist to record which instruction is allowed to access which memory address. It instruments all memory-accessing instructions to perform the check during runtime. DataShield [11] reserves a special memory region for critical variables to facilitate checks. Even though it concerns only selected data, DataShield still needs to instrument all memory-write instructions to prevent illegal access to the reserved memory region. In general, previous works on program data integrity share the same fundamental approach, which we call *address-based checking*. They need to check every memory-write instruction and determines if it should be allowed to write to the *referenced address*.

In contrast, CVI takes a new approach to data integrity checking, called Value-based Define-Use Check. It checks if the value of a critical variable at an instrumented load instruction (i.e., use) remains the same as the value recorded at the previous instrumented store instruction (i.e., define). At an instrumented store instruction, CVI makes a copy of the value in a secure region guarded by TrustZone. At an instrumented load instruction, CVI compares the value read from memory with the copy recorded in TrustZone. Any data corruption causes a value mismatch and therefore a CVI breach. CVI only needs to instrument the instructions that are supposed to read/write the critical variables whereas address-based checkers have to instrument all memory-write instructions even if only selected data needs checking. Table I shows the comparison between CVI and the previous data integrity checking techniques.

TABLE I: Comparison between Different Data Integrity Mechanisms (R:Read, W:Write)

Name	Instrumentation	Whitelist	Check
DFI [13]	All memory R&W	Yes	Addr.
WIT [3]	All memory W	Yes	Addr.
DataShield [11]	All memory W	No	Addr.
CVI	Critical variable R&W	No	Value

Critical Variable Identification & Expansion: OAT compiler automatically identifies *control-dependent variables*, including branch/loop condition variables. Note that we do not include code pointers as critical variables. This is because all control-flow violations, including those resulted from corrupted code pointers, are captured by the control-flow part of OEI attestation. OAT relies on programmers to annotate *semantically critical variables* because labeling such variables is subjective and specific to program semantics.

The control-dependent variables (automatically detected) and the semantically critical variables (annotated by programmers) form the initial set of critical variables. Our compiler then automatically expands this set to include the direct and indirect **pointers** to these variables and their **dependencies**.

Pointers to a critical variable allow indirect define or use of the variable. Therefore, we treat such data pointers as special critical variables, referred to as *critical pointers*. Our compiler iteratively identifies them from a global points-to map produced by the standard Anderson pointer analysis.

Dependencies of a critical variable `V` are the variables that may influence the value of `V`. Verifying the integrity of `V` implies verifying the integrity of both `V` and its dependencies. Given a set of critical variables, our compiler finds all their dependencies by first constructing the program dependence graph and then performing a backward traversal for each variable along the data dependence edges. Newly discovered variables during the traversal are added to the critical variable set. The iterative search for dependencies stops when the set reaches a fixpoint. This automatic dependency discovery also simplifies critical variable annotation: programmers only need to annotate one, not all, variable for each piece of critical data.

Value-based Define-Use Check: OAT compiler instruments all define- and use-sites for each variable in the expanded critical variable set. During runtime, the instrumentation at each define-site captures the identity of the critical variable (i.e., a compiler-generated label) and the value to be assigned to the variable. It sends the information to the measurement engine in the Secure World via the trampolines. Similarly, the use-site instrumentation captures the variable identity and the current value of the variable and sends them to the measurement engine.

For an array-typed critical variable, each array element access is instrumented by OAT compiler. OAT compiler identifies and instruments every memory access whose target address is calculated as an offset relative to a critical variable. This critical variable can be an array name or a pointer that points to a buffer. This design also uniformly covers multi-dimensional arrays and nested arrays because in both cases an array element access is based on a memory address that is calculated from the array's name or base address. From the measurement engine's perspective, it only sees critical variables or critical array elements identified by their addresses, rather than entire array objects. The integrity of each element is checked independently during runtime.

The measurement engine maintains a hashmap which stores pairs of `<VariableID, Value>`. $VariableID$ is the address of a variable or an array element. The measurement engine updates the hashmap at each instrumented define-site and checks the value at each instrumented use-site. Regardless whether a variable is on the stack (local) or the heap (global), its def-use sites should always have matching values. A value mismatch indicates a corrupted critical variable. The measurement engine finally sends the CVI checking result along with the control-flow measurements in a signed blob to the remote verifier.

Pointer-based Access to Critical Variables: Consider an example: `*(P+n) = x`, where `P` is the base address of a critical array `A`; `n` is the dynamic index. This is a legitimate critical variable define-site. Assuming that, due to a program bug, the dereference `*(P+n)` could go out of the bounds of `A` and reach another critical variable `B`, the measurement engine would then mistakenly update the value of `B` to `x` in its hashmap when this out-of-bounds write happens.

We solve this issue by enforcing dynamic bounds checking on critical pointers. When a critical pointer is dereferenced, the measurement engine checks if the pointer dereference breaches the bounds of the current pointee. This check relies on the dynamic bounds information (similar to SoftBound [46]) collected by the measurement engine for each critical pointer. If a bounds breach is found, the measurement engine performs CVI check only on the overlapping bytes between the accessed memory region and the initially pointed variable (i.e., it only updates or checks the value of the expected pointee). This design ensures CVI check correctness while allowing intentional out-of-bounds pointer dereferences in normal programs.

VI. Implementation

Our OAT prototype implementation includes *6,017* lines of C++ code for the compiler (an LLVM module), *440* lines of C and assembly code for the trampoline library, *476* lines of C code for the measurement engine, and *782* lines of Python code for the verification engine.

Hardware & Software Choices: We selected a widely used IoT development board, the HiKey board (ARM Cortex-A53), as our reference device for prototyping. We made this choice due to the board's unlocked/programmable TrustZone, its full compatibility with open-source TEE OS, and its reliable debugging tools. We do realize that the board has a relatively powerful processor compared to some low-end embedded devices. However, no development board currently available comes with a low-end ARM processor and an unlocked TrustZone at the same time. Nevertheless, OAT's design and implementation are neither specific to this board nor dependent on powerful processors. The only hardware requirement OAT has is the TrustZone extension, which is available on many commercial embedded ARM SoCs.

We used OP-TEE [45] as our TEE OS (the OS for the Secure World). Although OAT is designed for bare-metal embedded devices (i.e., no stand-alone OS in the Normal World), we used the vendor-customized Linux as the Normal World OS solely for the purpose of easily booting and debugging the board. OAT itself does not assume the presence of a Normal World OS. Even though OAT's performance can be negatively affected by the unnecessary OS, our evaluation (§VIII) shows the overhead under this disadvantaged configuration is still acceptable.

The implementation details of the OAT system is attached in Appendix §A.

VII. Discussion

Multithreading: Our current prototype only supports single-thread programs, which constitute the majority of today's embedded programs. To attest multi-thread programs, we need to augment the OAT compiler to instrument threading-related events and have the measurement engine collect measurements and perform checks on a per-thread basis. We consider multi-threading support out-of-scope for this paper because it does not need any change to the design of OEI attestation but requires significant engineering efforts to implement.

Interrupt Handling: Interrupt handling poses a challenge to the control flow verification due to its asynchronous nature. When an interrupt happens in the middle of an attested operation, we do not know in advance where the interrupted location is. If the measurement engine cannot recognize and process interrupts, they may introduce out-of-context control flow events that can fail or confuse the verification.

OAT overcomes this challenge by treating each interrupt handler invoked during an operation as an execution of a sub-program. It instruments the handler both at the entry and the exit points. The instrumentation notifies the measurement engine of the beginning and the end of such a sub-program. Thus the control flow events of the handler are recorded in a separate trace and hash and verified independently. OAT also checks if an invoked interrupt handler matches the interrupt that triggered the handler.

Annotation: OAT allows programmers to optionally annotate semantically critical variables for CVI attestation. However, this annotation is *not required* for detecting data-oriented programming [33], control-flow bending [10], or similar data-only attacks, whose target data (i.e., control-dependent variables) are automatically detected by OAT compiler and included in CVI verification. On the other hand, the annotation process

TABLE II: Runtime Overhead Breakdown

Overhead Sources	Time(CPU Cycles)	Time(ms)
Attestation Init (O_{at_init})	$5.1*10^7$	42.879
Trampoline Func(O_{tramp})	$5.5*10^4$	0.045
Attestation Exit (O_{at_exit})	$2.6*10^7$	21.351

for semantically critical variables is simple and facilitated by the compiler's automatic dependency analysis.

VIII. Evaluation and Analysis

We conducted: (*i*) micro performance tests to measure the overhead of each step in OEI attestation; (*ii*) macro performance tests on 5 real embedded programs of different kinds to examine OAT's overall overhead; (*iii*) tests against possible attacks; (*iv*) analysis on how OAT defends against evasions.

A. Micro Performance Tests

The runtime overhead of OEI attestation can be broken down into three parts, as shown in the first column in Table II: (*i*) attestation initialization (O_{at_init}), taking place at the entry of an attestation scope (i.e., an operation); (*ii*) trampoline invocation (O_{tramp}), including a direct `smc` call to initiate the world switch and a return from the Secure World (i.e., the roundtrip world-switch overhead); (*iii*) attestation exit (O_{at_exit}), happening at the end of an attestation scope.

We created a test program that incurs each type of overhead exactly once. We ran this program for 1,024 times and obtained the average overhead in terms of CPU cycles as well as time used, as shown in Table II. O_{at_init} and O_{at_exit} are orders of magnitudes larger than O_{tramp}. This is because they involve establishing or terminating the attestation session and the communication between the Normal and the Secure worlds. Since the initialization and exit happen only once for each attestation, their overhead tends to blend in the (much longer) operation execution time and is unnoticeable.

B. Tests on Real Embedded Programs

We selected 5 open-source embedded programs to evaluate the end-to-end overhead of OAT. We could not test more programs because of the non-trivial manual effort required for porting each program to our development board. This requirement is due to embedded programs' device-specific nature. It is not posed by our system. We picked these programs because they represent a reasonable level of variety. Some of them may seem toy-like, which is not intentional but reflects the fact that most embedded programs are simple by design. We note that these programs are by no means standard benchmarks. In fact, there are no standard benchmarks for bare-metal embedded devices available at the time of writing. It is also rare for embedded device vendors to publicly release their firmware in source code form. The 5 selected embedded programs are:

- *Syringe Pump* (`SP`) is a remotely controlled liquid-injection device, often used in healthcare and food processing settings. We apply OEI attestation to the "injection-upon-command" operation.
- *House Alarm System* (`HA`) [25] is an IoT device that, when user-specified conditions are met, takes a picture and triggers an alarm. We apply OEI attestation to its "check-then-alarm" operation.
- *Remote Movement Controller* (`RM`) [27] is an embedded device that allows the physical movement of its host to be controlled remotely, similar to the example given in §III-A. We attest the "receive-execute-command" operation.
- *Rover Controller* (`RC`) [28] controls the motor on a rover. We attest the "receive-execute-command" operation.
- *Light Controller* (`LC`) [26] is a smart lighting controller. We attest the "turn-on/off-upon-command" operation.

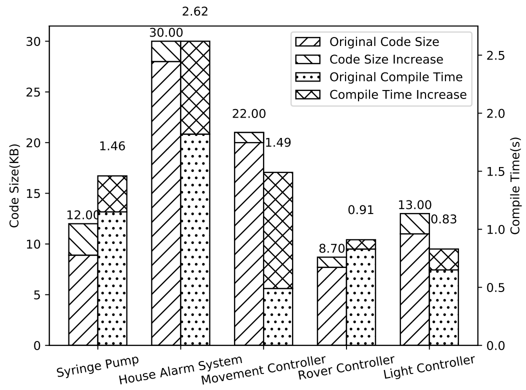


Fig. 3: Compile-time overhead

Compile-time Overhead: We instrumented OAT compiler to measure its own performance in terms of binary size increase and compilation delay. Figure 3 shows the results for each program. The absolute increase in code size ranges from 1 to 3 `KB`, which is acceptable even for embedded devices, despite the seemingly large percentage (13%). We find that the size increase is not proportional to the original code size and is dominated by the trampoline library, which has a fixed size.

The compilation delay caused by the attestation-related code analysis and instrumentation may seem high, averaging 62%. But in absolute terms, the delay is less than 1 second for the tested programs. We believe this is acceptable given that (*i*) the overhead is on par with similar code analysis and instrumentation techniques; (*ii*) the compilation is offline and happens only once for each program version.

Operation Execution Time & Instrumentation Statistics: For each test program, we measured the execution times with and without OEI attestation enabled for the selected operation. The results are shown in the column named "Operation Exec. Time" in Table III. The sub-column "Overhead" shows the relative delay caused by OAT to operation executions, averaging 2.7%. We observed that the delays are unnoticeable and blend in the much longer end-to-end execution time of the

TABLE III: Runtime overhead measured on 5 real embedded programs

Prog.	Operation Exec. Time			OAT Instrumentation Statistics					Blob	Verification
	w/o OEI (s)	w/ OEI (s)	Overhead (%)	B.Cond	Def-Use	Ret	Icall/Ijmp	Critical Var.	Size (B)	Time (s)
`SP`	10.19	10.38	1.9%	488	2	1946	1	20	69	5.6
`HA`	5.28	5.36	1.6%	147	91	33	2	6	44	0.61
`RM`	10.01	10.13	1.3%	901	100	100	100	7	913	1.74
`RC`	2.55	2.66	4.5%	14	33	1	1	8	10	0.13
`LC`	5.33	5.56	4.4%	931	2420	10	10	4	205	1.35
Avg.	N/A	N/A	2.7%	496	529	418	23	9	248	1.89

operations.

It is worth noting that the execution delay caused by our attestation may vary significantly as the following two factors change: the length/duration of the attested operation and the frequency of critical variable def-use events. For shorter operations, the attestation overhead tends to be higher percentage-wise. For instance, the operation in `RC` (the shortest among the tested programs) takes 2.55 seconds to finish without attestation and 2.66 seconds with attestation, resulting in the highest relative overhead (4.5%) among all tested programs. However, this does not mean the absolute delay, in this case, is longer than others or unacceptable.

The more frequent the def-use events of critical variables are, the higher the attestation delay becomes. For example, the operations in `HA` and `LC` are similar in lengths. But the attestation overhead on `HA` (1.3%) is lower than the overhead on `LC` (4.4%) partly because `HA` has fewer def-use events of critical variables.

In the "Instrumentation Statistics" column, we show, for each operation execution, the numbers of the instrumented events encountered during the attestation (including conditional branches, def-use checks, returns, and indirect calls/jumps) as well as the number of critical variables selected. These statistics provide some insights into the overhead reported earlier. The instrumented events occur about thousands of times during an operation, which translates roughly to an average per-operation delay of 0.15 seconds.

Measurement Engine Memory Footprint and Runtime Overhead: The measurement engine inside TEE consumes memory mainly for three purposes: the BLAKE-2s HASH calculation, the critical data define and use check, and the forward control-flow trace recording, including taken or not-taken bits and indirect jump/call destination. The BlAKE-2 HASH function only requires less than 2KB for storing the block buffer, 32 Bytes for the IV, 160 Bytes for the sigma array, and some temporary buffers. The critical data check requires a static HASH table of 4KB with 512 slots, and a dynamic pool for critical variables (the size of this pool is proportional to the number of critical variables; in our evaluation, the pool size is less than 2KB). The forward control-flow trace in our evaluation is no more than 2KB. The whole memory footprint of the measurement engine is less than 10KB for the real embedded applications used in our evaluation.

The runtime overhead of the measurement engine comes from three sources (i.e., the three major tasks of the measurement engine): calculating the hash upon function return events, recording critical data define events, and verifying critical data use event. In our evaluation, on average, processing one return event and calculating the new hash takes $0.19\,\mu$s; recording a critical data define event takes $11.04\,\mu$s; verifying a critical data use event takes $2.03\,\mu$s. Obviously, hash calculation is relatively fast whereas critical data event processing requires a longer time mainly because it involves hash table lookup or memory allocation for a new entry.

TABLE IV: Number of Instrumentation Sites: Value-based (R1) and Address-based (R2)

	SP	HA	RM	RC	LC	Avg.
R1	56	37	57	20	41	-
R2	140	388	842	45	131	-
R1 / R2	40%	9.5%	6.8%	44.4%	31.2%	26%

Value-based Check vs. Addressed-based Check: To show the performance difference between value-based checking (CVI) and address-based checking (e.g., DFI), we measured the number of instrumented instructions needed in both cases for all of the test programs. As shown in Table IV, on average, CVI's instrumentation is 74% less than the instrumentation required by address-based checking (i.e., a 74% reduction). Specifically, CVI's instrumentation is as little as 6.8% of what DFI requires when the program is relatively large (e.g., `RM`). The number only increases to 44.4% when we annotated most of the variables as critical in `RC`.

Space-efficiency of Hybrid Attestation: Our control-flow attestation uses the hybrid scheme consisting both forward traces and backward hashes to achieve not only complete verifiability but also space-efficiency. To quantify the space-efficiency, we compared the sizes of the control-flow traces produced by OAT (R1 in Table V) and the traces produced by pure trace-based CFI (R2 in Table V). On average, OAT's traces take only 2.24% of space as needed by control-flow traces (i.e., a 97% reduction). This result shows that our hybrid scheme is much better suited for embedded devices than solely trace-based CFI in terms of space efficiency.

On the other hand, compared with existing hash-based attestation schemes, OAT's attestation blobs are not of fixed-length and grow as attested operations execute, which may lead to overly large attestation blobs. However, in practice, OAT attestation blobs are reasonably small in size. Based

TABLE V: Control-flow Trace Size (Bytes): With Return Hash (R1) and Without Return Hash (R2)

	SP	HA	RM	RC	LC	Avg.
R1	69	44	913	10	205	-
R2	42941	3772	13713	585	13725	-
R1 / R2	0.2%	1.1%	6.7%	1.7%	1.5%	2.24%

on our experiments, the average blob size is $0.25kb$. The individual blob size for each program is shown in the "Blob Size" column in Table III. We attribute this optimal result to two design choices: (*i*) the hybrid measurement scheme that uses fixed length hashes for verifying returns (more frequent) and traces for verifying indirect forward control transfers (less frequent); (*ii*) the operation-scoped control-flow attestation, which generates per-operation measurements and is enabled only when an operation is being performed.

Verification Time: We also measured OAT verifier's execution time when it checks the attestation blob generated for each program. The result is shown in the "Verification Time" column in Table III, averaging 1.89 seconds per operation. It shows that verification is not only deterministic but fast. On average, the verification is one order of magnitude faster than the original execution (Table III). This result echoes that the verification is not a re-run of the program. It is a static abstract execution guided by the measurement stream.

C. Attack Detection via OEI Attestation

Due to the lack of publicly available exploits for bare-metal devices, we injected vulnerabilities to the previously discussed test programs, launch basic control-flow hijacks and data corruption, and examine if the measurements generated by OAT capture these attacks.

Specifically, we injected to the programs the vulnerabilities similar to those shown in Listing 1. We then exploited the vulnerabilities to (*i*) overwrite a function pointer; (*ii*) corrupt a critical variable; (*iii*) trigger an unintended operation. By verifying the measurements generated by OAT, we found that, in each test case, (*i*) the illegal control-flow transfer caused by the subverted function pointer is recorded in the measurement stream; (*ii*) the CVI flag is set due to the failed CVI check; (*iii*) the unintended operation is detected because the reconstructed code path does not match the requested operation.

Although these tests are simple and created by ourselves, they do demonstrate the basic functioning of our prototype and confirm OEI attestation as a viable way for remote verifiers to detect those attacks that are currently undetectable on embedded devices. Moreover, they showcase that IoT backend can now use OAT to remotely attest the operations performed by IoT devices and establish verifiable trust on these devices.

D. Security Analysis

Our threat model (§III-D) anticipates that attackers may find and exploit unknown vulnerabilities in the embedded programs running in the Normal World. However, we assume code injection or modification cannot happen, which the existing code integrity schemes for embedded devices already prevent [16], [37].

To evade OAT, a normal-world attacker would need to ① disable the instrumentation or the trampolines, ② abuse the interfaces that the measurement engine exposed to the trampolines, or CA-TA interfaces, ③ manipulate the control flow in a way to generate a HASH collision, thus bypassing the verification, or ④ modify the attestation blob including replay an old recorded blob.

① is ruled out by the code integrity assumption. Plus, attempts to divert the control-flows of instrumented code or trampolines are always recorded in the control-flow trace and detected later by the verifier. Our design prevents ② as follows. OAT compiler disallows world-switching instructions (`smc`) used outside of the trampoline library. This restriction ensures that only the trampoline functions can directly invoke the CA-TA interfaces and the rest of the code in the Normal World cannot. To further prevent code-reuse attacks (e.g., jumping to the interface invocation point in the library from outside), OAT loads the library in a designated memory region. The compiler masks the target of every indirect control transfer in the embedded program so that the trampoline library cannot be reached via indirect calls or jumps or returns (i.e., only hard-coded direct calls from each instrumentation site can reach trampolines). This masking-based technique is highly efficient and is commonly used for software fault isolation. As a result, ② is prevented.

As for ③, we assume that the attacker may exploit program vulnerabilities and manipulate the control flow of the program in arbitrary ways. We prove that (see Appendix B) our control-flow verification mechanism cannot be bypassed by such a powerful attacker. Our proof shows that bypassing our verification is at least as hard as finding a hash collision, which is practically infeasible considering that BLAKE-2s is as collision-resistent as SHA3 [7].

Our verification scheme prevents ④ because the integrity of the attestation blob is guarded by a signature generated from TEE with a hardware-provisioned private key. A verifier can easily check the signature and verifies the integrity of the attestation blob. Replay attack is also prevented by checking whether the cryptographic nonce inside the attestation blob matches what originally was generated by the verifier.

There is no higher privileged code (e.g., a standalone OS) that needs to be protected or trusted because OAT targets bare-metal embedded devices. For the same reason, it is realistic to require the firmware to be entirely built using OAT compiler.

IX. Related Work

Remote Attestation: Early works on remote attestation, such as [60][44], were focused on static code integrity, checking if code running on remote devices has been modified. A series of works [4], [50], [21], [38] studied the Root of Trust for remote attestation, relying on either software-based TCB or hardware-based TPM or PUF. Armknecht et al. [5] built a security framework for software attestation.

Other works went beyond static property attestation. Haldar et al. [61] proposed the verification of some high-level semantic properties for Java programs via an instrumented Java virtual machine. ReDAS [36] verified the dynamic system properties. Compared with our work, these previous systems were not designed to verify control-flow or dynamic data integrity. Further, their designs do not consider bare-metal embedded devices or IoT devices. Some recent remote attestation systems addressed other challenges. A tool called DARPA [35] is resilient to physical attacks. SEDA [6] proposed a swarm attestation scheme scalable to a large group of devices. In contrast, we propose a new remote attestation scheme to solve a different and open problem: IoT backend's inability to verify if IoT devices faithfully perform operations without being manipulated by advanced attacks (i.e., control-flow hijacks or data-only attacks). Our attestation centers around OEI, a new security property we formulated for bare-metal embedded devices. OEI is operation-oriented and entails both control-flow and critical data integrity.

A recent work called C-FLAT [2] is closely related to our work. It enabled control-flow attestation for embedded devices. However, it suffers from unverifiable hashes, especially when attested programs have nested loops and branches. This is because verifying a control-flow hash produced by C-FLAT requires the knowledge of all legitimate control-flow hashes, which are impossible to completely pre-compute due to the unbounded number of code paths in regular programs (i.e., the path explosion problem). In comparison, OAT uses a new hybrid scheme for attesting control-flows, which allows deterministic and fast verification. Moreover, OAT verifies Critical Variable Integrity and can detect data-only attacks, which C-FLAT and other previous works cannot.

Online CFI Enforcement: Although control-flow attestation has not been well investigated, CFI enforcement is a topic that has attracted a rich body of works since its debut [1]. A common goal shared by many CFI enforcement methods such as [58], [49], [63], [48], [17], [18] is to find a practical trade-off between runtime overhead and the level of precision. Previous works such as [65], [64] also introduced CFI enforcement to COTS or legacy binaries. CPI [41] prevents control flow hijacking by protecting code pointers in safe regions. These work made CFI increasingly practical for adoption in the real world and serves as an effective software exploitation prevention mechanism.

However, enforcing fine-grained CFI and backward-edge integrity can be still too heavy or impractical for embedded devices, mostly because of the limited CPU and storage on such devices. Apart from less demanding on hardware resources, OEI attestation has another advantage over online CFI enforcement: it allows remote verifiers to reconstruct the exact code paths executed during an operation, which enables full CFI checking (as opposed to a reduced or coarse-grained version) as well as other postmortem security analysis.

Moreover, CFI enforcement is not enough when it comes to data-only attacks, such as control-flow bending, data-oriented programming, etc. [14], [10], [33], [32]. But these attacks do violate OEI and can be detected by OAT.

Runtime Data Protection: A series of work addressed the problem of program data corruption via dynamic bounds checking [20], [46], [19] and temporal safety [47]. DFI [13] and WIT [3] took a different approach. They use static analysis to derive a policy table specifying which memory addresses each instruction can write to. They instrument all memory-access instructions to ensure the policy is not violated during runtime. Although effective at preventing data corruption, these techniques tend to incur high runtime overhead due to the need to intercept and check a large number of memory accesses. We refer to this line of work as address-based checking. In contrast, we define Critical Variable Integrity and use the new Value-based Define-Use Check to verify CVI. Our check is selective (i.e., it only applies to critical variables) and lightweight. It is value-based and does not require complex policies or extensive instrumentation. The CVI verification and the control-flow attestation mutually compensate each other, forming the basis for OEI verification.

DataShield[11] applies selective protection to sensitive data. Their definition of sensitive data is type-based and also needs programmer annotation. It relies on a protected memory region to isolate the sensitive data and performs address-based checking. In comparison, our critical variable annotation is more flexible and partly automated. Instead of creating designated safe memory regions, which can be unaffordable or unsupported on embedded devices, we perform lightweight value-based checks. Unlike DataShield, OEI does not concern data confidentiality.

X. Conclusion

We tackle the open problem that IoT backends are unable to remotely detect advanced attacks targeting IoT devices. These attacks compromise control-flow or critical data of a device, and in turn, manipulate IoT backends. We propose OEI, a new security property for remotely attesting the integrity of operations performed by embedded devices. OEI entails *operation-scoped CFI* and *Critical Variable Integrity*.

We present an end-to-end system called OAT that realizes OEI attestation on ARM-based bare-metal embedded devices. OAT solves two research challenges associated with attesting dynamic control and data properties: incomplete verification of CFI and heavy data integrity checking. First, OAT combines forward-edge traces and backward-edge hashes as control-flow measurements. It allows fast and complete control-flow verification and reconstruction while keeping the measurements compact. Second, OAT enforces selective value-based variable integrity checking. The mechanism is lightweight thanks to the significantly reduced instrumentation. It enables the detection of data-only attacks for the first time on embedded devices. It allows IoT backends to establish trust on incoming data and requests from IoT devices.

References

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene 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, pages 743–754, New York, NY, USA, 2016. ACM.

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 263–277. IEEE, 2008.

[4] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, SP '97, pages 65–, Washington, DC, USA, 1997. IEEE Computer Society.

[5] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. A security framework for the analysis and design of software attestation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 1–12, New York, NY, USA, 2013. ACM.

[6] N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. Seda: Scalable embedded device attestation. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 964–975, New York, NY, USA, 2015. ACM.

[7] BLAKE2. Blake2 — fast secure hashing. https://blake2.net/, 2019.

[8] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: Generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 27–38. ACM, 2008.

[9] Capstone. Capstone disassembly framework. https://www.capstone-engine.org/.

[10] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *USENIX Security Symposium*, pages 161–176, 2015.

[11] Scott A Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 193–204. ACM, 2017.

[12] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 400–409, New York, NY, USA, 2009. ACM.

[13] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 147–160, Berkeley, CA, USA, 2006. USENIX Association.

[14] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 12–12, Berkeley, CA, USA, 2005. USENIX Association.

[15] Shuo Chen, Jun Xu, Emre Can Sezer, Prachi Gauriar, and Ravishankar K Iyer. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, volume 14, 2005.

[16] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *IEEE Symp. on Security and Privacy. IEEE*, 2017.

[17] John Criswell, Nathan Dautenhahn, and Vikram Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 292–307. IEEE, 2014.

[18] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *NDSS*, volume 26, pages 27–40, 2012.

[19] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. Hardbound: architectural support for spatial safety of the c programming language. In *ACM SIGARCH Computer Architecture News*, volume 36, pages 103–114. ACM, 2008.

[20] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the 28th international conference on Software engineering*, pages 162–171. ACM, 2006.

[21] Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik. SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust. In *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*, San Diego, UNITED STATES, 02 2012.

[22] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6), 2011.

[23] FDA. Cybersecurity vulnerabilities identified in implantable cardiac pacemaker, August 2017.

[24] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding control flows using intel processor trace. *ACM SIGOPS Operating Systems Review*, 51(2):585–598, 2017.

[25] Github. House Alarm System. https://github.com/ddrazir/alarm4pi.

[26] Github. Light Controller. https://github.com/Barro/light-controller.

[27] Github. Remote Movement Controller. https://github.com/bskari/pi-rc/tree/pi2.

[28] Github. Rover Controller. http://github.com/Gwaltrip/RoverPi/tree/master/tcpRover.

[29] GlobalPlatform. GlobalPlatform TEE Specifications. https://www.globalplatform.org/specificationsdevice.asp.

[30] Andy Greenberg. Hacker says he can hijack a $35 k police drone a mile away, 2016.

[31] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 461–472. ACM, 2016.

[32] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 177–192, Berkeley, CA, USA, 2015. USENIX Association.

[33] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 969–986. IEEE, 2016.

[34] Troy Hunt. Controlling vehicle features of nissan leafs across the globe via vulnerable apis. https://www.troyhunt.com/controlling-vehicle-features-of-nissan/, February 2016.

[35] Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. Darpa: Device attestation resilient to physical attacks. In *Proceedings of the 9th ACM Conference on Security &#38; Privacy in Wireless and Mobile Networks*, WiSec '16, pages 171–182, New York, NY, USA, 2016. ACM.

[36] Chongkyung Kil, Emre C. Sezer, Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. Remote attestation to dynamic system properties: Towards providing complete system integrity evidence. *Dependable Systems & Networks, 2009.*, 2009.

[37] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *NDSS*, 2018.

[38] Joonho Kong, Farinaz Koushanfar, Praveen K. Pendyala, Ahmad-Reza Sadeghi, and Christian Wachsmann. PUFatt: Embedded Platform Attestation Based on Novel Processor-Based PUFs. In *DAC*, page 6. ACM, 2014.

[39] Tim Kornau. *Return oriented programming for the ARM architecture*. PhD thesis, Master's thesis, Ruhr-Universität Bochum, 2010.

[40] Brian Krebs. Reaper: Calm before the iot security storm. https://krebsonsecurity.com/2017/10/reaper-calm-before-the-iot-security-storm/, October 2017.

[41] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 147–163, Berkeley, CA, USA, 2014. USENIX Association.

[42] Keen Security Lab. New car hacking research: Tesla motors. http://keenlab.tencent.com/en/2017/07/27/New-Car-Hacking-Research-2017-Remote-Attack-Tesla-Motors-Again/, 2017.

[43] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[44] Yanlin Li, Jonathan M. McCune, and Adrian Perrig. VIPER: verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 3–16, 2011.

[45] Linaro. OP-TEE. https://www.op-tee.org.

[46] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[47] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[48] Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 199–210. ACM, 2013.

[49] Ben Niu and Gang Tan. Per-input control-flow integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 914–926, New York, NY, USA, 2015. ACM.

[50] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 414–429, Washington, DC, USA, 2010. IEEE Computer Society.

[51] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero. An experimental security analysis of an industrial robot controller. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 268–286, May 2017.

[52] Rapid7. Multiple vulnerabilities in animas onetouch ping insulin pump. https://blog.rapid7.com/2016/10/04/r7-2016-07-multiple-vulnerabilities-in-animas-onetouch-ping-insulin-pump/, October 2016.

[53] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin OFlynn. Iot goes nuclear: Creating a zigbee chain reaction. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 195–212. IEEE, 2017.

[54] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282, May 2004.

[55] Arvind Seshadri, Mark Luk, Elaine Shi, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. *SIGOPS Oper. Syst. Rev.*, 39(5):1–16, October 2005.

[56] Tom Spring. New mirai variant carries out 54-hour ddos attacks. https://threatpost.com/new-mirai-variant-carries-out-54a-hour-ddos-attacks/124660/, March 2017.

[57] Christos Stergiou, Kostas E Psannis, Byung-Gyu Kim, and Brij Gupta. Secure integration of iot and cloud computing. *Future Generation Computer Systems*, 78:964–975, 2018.

[58] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc &#38; llvm. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 941–955, Berkeley, CA, USA, 2014. USENIX Association.

[59] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical context-sensitive cfi. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 927–940. ACM, 2015.

[60] Arvind Seshadri Adrian Perrig Leendert van Doorn Pradeep Khosla. Using software-based attestation for verifying embedded systems in cars. *S&P, Oakland*, 2004.

[61] Deepak Chandra Vivek Haldar and Michael Franz. Semantic remote attestation – a virtual machine directed approach to trusted computing. *VM*, 2004.

[62] Jacob Wurm, Khoa Hoang, Orlando Arias, Ahmad-Reza Sadeghi, and Yier Jin. Security analysis on consumer and industrial iot devices. In *Design Automation Conference (ASP-DAC), 2016 21st Asia and South Pacific*, pages 519–524. IEEE, 2016.

[63] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 29–40. ACM, 2011.

TABLE VI: Instrumented Instructions for Control and Data Measurement

Inst Type	Layer	Inst	Info to Record
Ind. Call	Assm.	`blr xr`	`xr`
Ind. Jump	Assm.	`br xr`	`xr`
Cond. Jump	Assm.	`b.cond,cbz` `cbnz,tbz,tbnz`	`true/false`
Data Access	IR	`load/store`	`addr,value`
Return	Assm.	`ret`	`pc,lr`

[64] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical control flow integrity and randomization for binary executables. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 559–573. IEEE, 2013.

[65] Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *USENIX Security Symposium*, pages 337–352, 2013.

Appendix A
Implementation Details of OAT System

Compiler-based Instrumentation: OAT compiler is built on LLVM [43]. Besides the typical compilation tasks, it performs (*i*) the analysis for identifying critical variables; (*ii*) the code instrumentation for collecting runtime measurements on control flow and critical variables. The analysis works on the LLVM IR. It first constructs the initial set of critical variables by traversing the IR and searching for control-dependent variables and programmer-annotated semantically critical variables. It then uses a field-sensitive context-insensitive Anderson pointer analysis to generate the global points-to information. The compiler uses this point-to information to recursively identify direct and indirect pointers to the critical variables (i.e., critical pointers). It also performs a backward slicing for each critical variable on the program dependence graph to find its dependencies. All critical pointers and dependencies are iteratively added to the critical variable set.

OAT compiler instruments the code via an added backend pass in LLVM. The instrumentation is needed at both the assembly level (for control-related instructions) and the IR level (for data-related instructions). This is important because the translation from the IR to the machine code can generate additional control-flow instructions that need to be instrumented. The compiler inserts calls to trampolines at instructions that can change the control flow of attested operations or store/load critical variables (Table VI). Though seemingly straightforward, this instrumentation, if not designed carefully, can break the original program because a trampoline call may corrupt the destination register used by the original control transfer. To avoid such issues, the instrumentation saves to the stack the registers that are to be used for passing the parameters to the trampoline. Moreover, the trampoline is responsible for handling the caller-saved registers (normally handled by callers rather than callees). This design reduces the number of inserted instructions at each instrumentation site. It also minimizes the stack frame growth. As a result, registers changed during a trampoline call are restored immediately after the call returns.

Measurement Engine: We built the measurement engine as a Trusted Application (TA) running in the Secure World. It handles events generated by the trampolines (i.e., the Client Application, or CA) during runtime. Control-flow events are only generated and handled during an active attestation window (when an attestation-enabled operation is executing). Internally, the measurement engine maintains, for each active operation, a *binary trace* (S_{bin}) for branches, an *address trace* (S_{addr}) for indirect calls/jumps, and a *hash* (H) for returns. At the end of an attestation session, the engine concatenates S_{bin} and S_{addr} to form a single measurement stream, S, in a sequentially parsable format: $Size(S_{addr})|S_{addr}|Size(S_{bin})|S_{bin}$.

Data load/store events are only triggered by critical variables. To perform CVI check, the engine maintains a hashmap to keep track of each critical variable's last-defined value. At every use-site of a critical variable, the engine checks if the observed value equals the stored value in the hashmap. If a mismatch is encountered, the engine sets a global flag, F, to indicate the CVI violation. If a violation is detected, the engine also records the variable address and the previous return address as the context information C, which allows the remote verifier to investigate the violation. Finally, the engine generates a signed attestation blob that consists of S, H, F, and C if CVI verification failed, along with a nonce N sent from the verifier who initiated the attestation. It will be passed back to the normal world who will finally send the signed attestation blob to our verification engine via the network. Although we use the normal world's network stack, we do not need to trust it. Any corruption of the blob is detectable by verifying the signature. Any denial of service by the normal world network stack also causes attestation failure.

CA-TA Interaction: We implemented three CA-TA communication interfaces compliant with the GlobalPlatform's TEE specification [29], a de-facto standard for TEE interface design. The interfaces are `oei_attest_begin`, `commit_event`, and `oei_attest_end`, used by the CA to notify the TA of the respective event. To prevent potential abuse (e.g., calling them via ROP), the measurement engine ensures that the interfaces can only be called by the trampolines and can never be invoked via indirect calls, jumps, or returns (details in §VIII-D).

Verification Engine: We prototyped a simple verification engine based on the Capstone disassembler [9]. It takes as input an attestation blob, the binary code that performed the operation under attestation, and a CFG extracted at compile time for that operation code. As described in §IV, the verification process is fairly straightforward, thanks to our hybrid measurement scheme.

For control-flow attestation, the verifier performs a static abstract execution of the disassembled binary code. This abstract execution is guided by the forward-edge traces in the attestation blob. It simply traverses through the code and performs CFI checks at each indirect control-flow transfer. It also simulates a call stack for keeping track of return addresses and updating the return hash, which is checked against the reported hash in the end. This abstract-execution-based verification is fast because it does not actually run the code or have to exhaustively explore all possible code paths. Moreover, unlike traditional attestation, which only gives a binary result, our verification allows for the reconstruction of the execution traces, which are valuable to postmortem analysis. For CVI verification, the verifier checks if the CVI violation bit is set in the attestation blog. If positive, it fails the attestation and outputs the context information.

Appendix B
Proof of Control-flow Verification

Let $h : \{0,1\}^{\star} \rightarrow \{0,1\}$ be a collision resistant hash function. Using h we can construct another hash function $H[op]$ which takes the sequence of values $\langle z_1, \cdots, z_m \rangle$ as follows: $H_1 = h(z_1)$ and $H_{i+1} = op(H_i, z_{i+1})$ $(1 \leq i < m)$. The value of H on the sequence is H_m. We rely on the following property, which restricts the binary operation op we can use.

> If h is collision resistant, then $H[op]$ is collision resistant

Our implementation uses BLAKE-2s as h and supports a variety of binary operations, such as concatenation and xor. Recall that if op is concatenation, $H[op]$ is very similar to the classic Merkle-Damgrad construction. For the rest of the note, we will fix the binary operation op (*i.e.,* we use concatenation as op in our implementation) and just write H instead of $H[op]$.

Let P be the program under consideration. Let $C(P) = \{c_1, \cdots, c_k\}$ and $R(P) = \{r_1, \cdots, r_k\}$ be the call and return sites in a program P (we will assume that the return site r_i corresponds to the call site c_i). Recall that a proof σ has three components (α, v, β), where $\alpha \in C(P)^*$ (a sequence of call sites), v is the hash value of the sequence of returns, and β is a sequence of jumps (direct and indirect) and conditional branches (essentially β has everything related to control-flow transfers, except calls and returns). A path π through the control-flow graph (CFG) program P is called *legal* if it satisfies two conditions: (A) the call and returns in π are balanced [4], (B) the jumps and conditional branches are legal (this can be easily checked by looking at the source code of P and the data values corresponding to the targets of the indirect jumps). $\Pi(P)$ denotes the set of execution paths through the CFG of the program P. The proof corresponding to an execution path $\pi \in \Pi(P)$ is denoted by $\sigma(\pi)$. Next, we describe our verification algorithm.

Verification. Our verifier $vrfy(P, \sigma)$ (let $\sigma = (\alpha, v, \beta)$) and is conjunction of two verifiers $vrfy_j$ and $vrfy_c$ described below.

- Verifier $vrfy_j(P, \sigma)$ checks that the jumps and branches in β are valid (i.e. this can be easily done because the verifier has the program P and the data values

[4]This means that call and returns satisfy the grammar with the following rules: $S \rightarrow c_i \; S \; r_i$ (for $1 \leq i \leq k$) and $S \rightarrow \epsilon$.

corresponding to the targets of the indirect jumps). If the jumps are valid, then $vrfy_j(P, \sigma) = 1$; otherwise $vrfy_j(P, \sigma) = 0$

- Verifier $vrfy_c(P, \sigma)$ checks the validity of calls and returns. This part is a bit more involved. Essentially $vrfy_c$ "mimics" how the hash of the returns are computed and then checks if the computed hash value matches the one in the proof σ. Verifier $vrfy_c$ maintains an auxiliary stack st and processes the sequence of calls $\alpha = \langle c_{j_1}, \cdots, c_{j_n} \rangle$ as follows: The calls in α are processed in order, and the verifier keeps running hash. Let us say we have processed $c_{j_1}, \cdots, c_{j_r}$ and are processing $c_{j_{r+1}}$. Recall that from the call site we can tell if there was a context switch in the program execution (a context switch means that we are executing in a different function). The call site has the location of the program, so we can inspect whether we are in the same function as the top of the stack (i.e., the location of $c_{j_{r+1}}$ is different from the location of the call site on top of the stack). If there was no context switch, then we push $c_{j_{r+1}}$ on the stack. If there was a context switch, then we pop the top of the stack (say c_u), compute $v' = op(v', h(r_u))$, and push $c_{j_{r+1}}$ on the stack. If r_u was the first return computed, then $v' = h(r_u)$. After all the calls have been processed, let the hash value be v'. The verifier $vrfy_c$ outputs a 1 if $v = v'$; otherwise it outputs a 0.

The verifier $vrfy(P, \sigma)$ is $vrfy_j(P, \sigma) \wedge vrfy_c(P, \sigma)$.

Definition 1. A proof σ is called *ambiguous* iff there are two paths π and π' and π such that: (I) $\sigma = \sigma(\pi) = \sigma(\pi')$ (II) π is legal and π' is illegal.

Note that if the verifier gets an ambiguous proof σ, then it cannot reject it because it could also correspond to a legal path π. Therefore, an adversary is free to take an illegal path π' corresponding to the ambiguous proof. Therefore, adversary's goal is to generate an ambiguous proof.

Essentially the lemma given below informally states that *if an adversary can generate an ambiguous proof, then they can find a collision in the hash function* $H[op]$. Hence, if $H[op]$ is collision resistant, then it will be hard for an adversary to find an ambiguous proof and "fool" the verifier.

Lemma 2. If there exists an ambiguous proof σ, then there is a collision in the hash function $H[op]$.

Proof: Let π and π' be two execution paths that result in the same proof σ. Moreover, let π be legal and π' be illegal (recall that σ is an ambiguous proof). Let r_π and $r_{\pi'}$ be the sequence of returns for the two paths π and π'. The set of direct jumps and call sequences for the two paths are the same (since they correspond to the same proof σ), so the sequence of returns has to be different (otherwise the two paths will be the same, which is a contradiction). However, the two sequences of returns hash to the same value under $H[op]$ because the paths correspond to the same proof. Thus, we have found a collision in $H[op]$. $\square$