DFA: Data Flow Attestation for Embedded Systems Software (Proposal)

CS 295 Spring 2020
UC Irvine

Sashidhar Jakkamsetti
sjakkams@uci.edu

Siddharth Narasimhan
siddhan@uci.edu

1. Problem

With the rapid increase and wide adoption of IoT devices, it has been critical to protect them from advanced software attacks. For this end, remote attestation (RA) comes in handy to detect the presence of malware on these devices which are generally deployed far in the field. There have been many schemes on remote attestation in the past decade, like SMART, VRASED, Intel SGX, Intel TrustLite and ARM TrustZone. However, all of these provide a platform for attestation of the state of the devices at a particular point of time, but none of them are capable enough to detect if the control flow execution of the device software has been hijacked. Attackers can easily send incorrect inputs to the IoT device and cause a buffer overflow leading to ROP attacks or data corruption attacks. C-FLAT is the first work that is done addressing such control flow attacks. But, C-FLAT does not do well against attacks which corrupt the execution of the program by changing the data pointers and their values without changing the control flow. In this project we are trying to propose a novel approach to attest the data flow of the untrusted program. In order to give context of such attacks and why it is important to detect them, let's consider the example of an open syringe pump (as shown in Figure 1), which is being controlled by an embedded device. These open syringe pumps are designed to inject critical drugs into a person's body, which makes the room for error very narrow. If an attacker hijacks the program execution and alters the amount of liquid to be injected per step, it could lead to fatal results. While this is just one scenario which sheds light on the importance of detecting program execution attacks on embedded systems software, the use cases for this are endless.

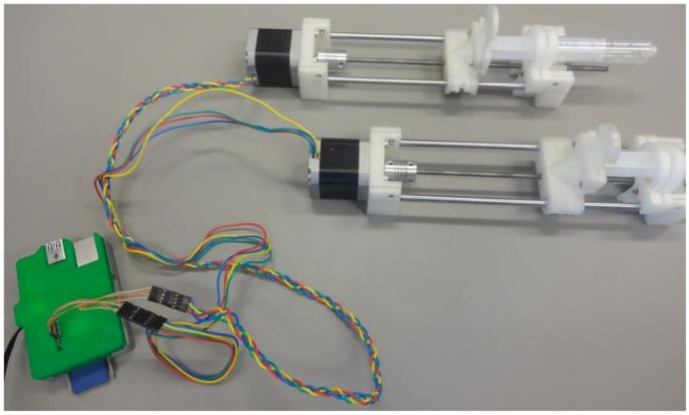

Figure 1: An open syringe pump

2. Context

Control Flow Integrity and Data Flow Integrity techniques can be a good solution to prevent such attacks. But, considering the low real estate and low budget that these IoT devices rely on, it is not a feasible solution to use them. Instead, it is reasonable to make these devices capable enough to send a report of execution of the untrusted program after every work request.

Most important related work for our project is C-FLAT. It proposes a novel scheme for attestation of the applications control flow path. C-FLAT complements static attestation by measuring the program's execution path at binary level, capturing its runtime behavior. In the technique proposed in C-FLAT, the prover computes an aggregated authenticator of the application's control flow path (sequence of executed instructions). This authenticator represents a fingerprint of the control flow path. It allows the verifier to trace the exact execution path in order to determine whether application's control flow has been compromised. However, in case an attacker manages to launch a buffer overflow to tamper with the data pointers or data itself during execution, and at the same time ensuring that the control flow graph does not change, then in this case C-FLAT will not be able to detect the attack. In embedded systems such as the open syringe pump described earlier, even the slightest change in the output may result in fatal results. This is a very important reason why there needs to be a system in place which detects any changes to the data pointers in the application. This problem leads to our next reference papers, "Securing software by enforcing data-flow integrity" and "DataShield". These papers suggest simple techniques that prevent data flow attacks by enforcing data flow integrity. One of them compute a data flow graph using static analysis, and it instruments the program to ensure that def-use chain integrity of each data variable is maintained as expected.

Example of data flow attack: Below (in Figure 2), we show a vulnerable piece of code, where the authenticated variable contains critical data which gives access to privileged parts of the program while execution. If the attacker can cause buffer overflow at line 5 in PacketRead by giving incorrect 'packet' to the program, then he can potentially surpass the if statement at line 7 and make the authenticated variable '1' gaining access to the privileged program.

```
1: int authenticated = 0;
2: char packet[1000];
3:
4: while (!authenticated) {
5:   PacketRead(packet);
6:
7:   if (Authenticate(packet))
8:     authenticated = 1;
9: }
10:
11: if (authenticated)
12:   ProcessPacket(packet);
```

Figure 2: Example of a code snippet vulnerable to data flow attacks.

3. Approach

We propose to design and implement a novel technique based on remote attestation architecture which checks the integrity of the data flow of the untrusted program running in a remotely deployed IoT device. Similar to C-FLAT, we will be making use of ARM-TrustZone-M as our trust anchor, because it is easily available to us through Raspberry Pi 3. In our approach (Figure 3 gives the architecture of our approach), we plan to incorporate a Secure Data Flow Attestation (DFA) monitor which resides in the ARM-TrustZone, which keeps track of the state of data and its address pointers of the untrusted program during runtime. After the execution, the Remote Attestation (RA) module will take the data flow report from the Secure DFA monitor and HMAC it with the device's key to send both the report and its MAC to the verifier.

It is not a scalable idea to attest the integrity of all the data variables in the untrusted program, so we are planning to find out the variables which potentially influence the output of the execution and apply attestation only on them. This requires Taint Analysis of the output data variables to find out which all data variables are important for execution and their other dependent variables. We call these data variables, **prime data variables** (*'steps'* data variable in line 4 of the program in the Figure 3 is a prime data variable) and their dependencies captured in **prime variable dominator graph**. In order to ensure the attestation of these prime data variables, we are planning to equip the DFA monitor with two major submodules. Firstly, a submodule, **DefUse Value Checker**, to maintain the def-use chains of each critical data variable, which logs the values of these variables at definition time and use time. Second submodule, **Prime Pointer Checker,** to log the data pointers (address pointers) of these variables during execution. By the help of these two submodules, the Secure DFA monitor can capture the 'value' and 'address point' behavior of all prime data variables and create a runtime report. On the verifiers end, based on the input it provided to the remote device, it will simulate the untrusted program execution locally and check whether the values and address pointers of the prime data variables are intact.

We are planning to implement this Secure DFA monitor on the device side and Verification module on the verifier's side. The Remote Attestation module is already present in the C-FLAT implementation, so we are planning to reuse it. This Secure DFA monitor and the Remote Attestation module will be deployed on Raspberry Pi 3. Verifier will be our laptops containing the Verification module. For communication we will be using a USB to UART converter like Waveshare DVK152.

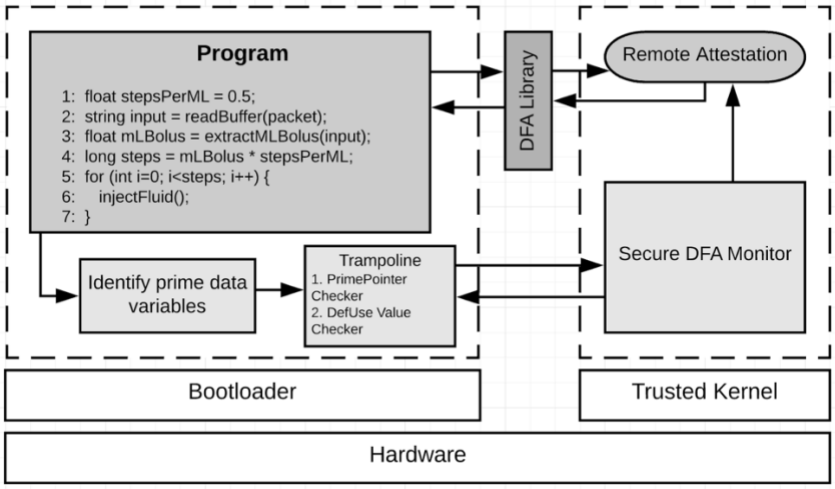

Figure 3: High Level architecture of DFA. Here 'steps' in line 4 of the program is a prime data variable and 'stepsPerML' is captured in the prime variable dependency graph.

4. Evaluation

We aim to deploy our design and prototype implementation of DFA to the Open syringe pump (shown in Figure 1). We are planning to evaluate our design on the following four aspects:

- **Program runtime performance overhead**, which is how much hindrance this DFA monitor is causing for the execution of the program on the device.
- **Verifier performance** on verification of the received report and simulating the program execution with the sent input.
- **Static code analysis and instrumentation overhead** (if required) for identifying the prime data variables.
- **Network bandwidth overhead**, because we are sending the report and the HMAC, which might be large.

For testing the DFA approach, we will introduce one or two simple data flow attacks, where data pointer or data value is altered and check if DFA is able to detect them.

5. Results

By the end of the quarter we aim to achieve the following deliverables:

- Design a Secure DFA monitor for attesting untrusted software on embedded systems.
- Deploy the implementation on a Raspberry Pi with Open Syringe Pump binary in it.
- Demonstrate our approach on real-time pure data attacks.
- A USENIX conference style 10-page report which includes the details of design, implementation and evaluation.

6. Timeline

- *Week of May 4:* Reproduce C-FLAT on a Raspberry Pi. Write a data flow attack to work on further. Taint analysis to find out prime data variables and prime variable dependency graph. Possible instrumentation if needed (and have enough time).
- *Week of May 11:* Design and implement submodule Prime Pointer Checker.
- *Week of May 18:* Design and implement submodule DefUse Value Checker.
- *Week of June 1:* Prepare for In-Class presentation. Perform experiments.
- *Week of June 8:* Write a USENIX conference style 10-page report and submit on June 12.