On Code Execution Tracking via Power Side-Channel

Yannan Liu[12], Lingxiao Wei[1], Zhe Zhou[3], Kehuan Zhang[3], Wenyuan Xu[4] and Qiang Xu[12]

[1]CUhk REliable Computing Laboratory (CURE)
[2] CUHK Shenzhen Research Institute
[123]The Chinese University of Hong Kong
[4] Department of Electronic Engineering, Zhejiang University
[1]{ynliu, lxwei, qxu}@cse.cuhk.edu.hk,[3]{zz113,khzhang}@ie.cuhk.edu.hk,[4]xuwenyuan@zju.edu.cn

ABSTRACT

With the proliferation of Internet of Things, there is a growing interest in embedded system attacks, e.g., key extraction attacks and firmware modification attacks. Code execution tracking, as the first step to locate vulnerable instruction pieces for key extraction attacks and to conduct control-flow integrity checking against firmware modification attacks, is therefore of great value. Because embedded systems, especially legacy embedded systems, have limited resources and may not support software or hardware update, it is important to design low-cost code execution tracking methods that require as little system modification as possible. In this work, we propose a non-intrusive code execution tracking solution via power-side channel, wherein we represent the code execution and its power consumption with a revised hidden Markov model and recover the most likely executed instruction sequence with a revised Viterbi algorithm. By observing the power consumption of the microcontroller unit during execution, we are able to recover the program execution flow with a high accuracy and detect abnormal code execution behavior even when only a single instruction is modified.

1. INTRODUCTION

Embedded devices controlled by microcontroller units are deployed everywhere. They are not only widely spread in our daily life with the proliferation of Internet of Things (IoT), but also extensively used in the global IT environments and critical infrastructures. Consequently, there is a growing interest in embedded system attacks and defense mechanisms. What makes both, especially defense, difficult is the limited capability of code execution monitoring on embedded systems, mainly caused by limited I/O interfaces and constrained-resources. This situation is unlikely to be alleviated any time soon by adding extract features, since updating embedded systems, especially legacy systems, is hindered due to safety or cost concerns. Thus, in the paper, we design a method for code execution tracking of embedded systems without requiring software or hardware modification. Such a method enables us to answer two important security related questions: (1) At a given time, which instruction in a code is being executed? (2) Given a source code, has it been modified and is the microcontroller unit (MCU) executing a malicious code?

Here, we illustrate how to utilize code execution tracking with two examples, but its applications are not limited to these two. (1) Locate the vulnerable code section for extracting private information of a system during execution. For instance, key extraction attacks [1, 2] assume that adversaries are aware of the code of the cryptographic algorithms. They analyze the source code to find vulnerable code sections, and locate these code sections during execution for private information extraction. Typically, prior work assumes locating code sections during execution is achievable and focus on the code analysis part. Our work fills in the blank. (2) Detect attacks that intend to hijack MCU's control-flow to execute malicious code [3–5]. One effective countermeasure to these attacks is to enforce control-flow integrity (CFI) [6], which tracks code execution and prevents code execution deviating from the control-flow graph (CFG) of the program. Over the last decade, a large number of CFI techniques [7–12] have been proposed. These techniques, despite their effectiveness, are inapplicable for many embedded systems, because the imposed overhead will overwhelm the resource-constrained devices and they typically require software and/or hardware modification, which is impossible for most embedded devices, especially legacy devices. Our work enables to apply CFI on embedded systems.

Code execution tracking via power side-channel is promising yet challenging. The advantage is that power side-channel leaks information about instructions being executed on MCU and obtaining such information needs no modification on the MCU itself. However, power measurement traces are quite noisy and it is difficult to extract useful information out of them. Prior work on recovering the type of executed instruction in a MCU via power-side channel [13] showed a rather low accuracy (about 60% in the best case). We manage to track code execution at a much higher accuracy by leveraging the control transfer information from CFG and using frequency analysis to reduce the noise in power side-channel. To be specific, the main contributions of this work include the followings.

- We propose to recover the instruction sequence by hidden Markov model (HMM). To increase the identification accuracy, we take advantage of the fact that for a given program, instructions should be executed in sequences obeying CFG, and identifying these sequences

CCS'16, October 24-28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978299

can increase the noise resilience than identifying every single instruction independently. Thus, we represent the CFG as the state machine in HMM. To efficiently utilize control transfer information from CFG, we use basic blocks in CFG as states in HMM. Correspondingly, we revise the classic HMM and Viterbi algorithm to cope with the challenges that basic blocks contain different numbers of instructions and hence have different lengths.

- We propose signal extraction techniques to design the observation symbols in HMM. By extracting high quality signals from power measurement traces, the impact of power measurement noises is dramatically reduced and we are able to further improve instruction recognition accuracy.
- We apply our proposed code execution tracking techniques for control-flow integrity checking, i.e., we obtain the likelihood of the reported instruction sequence from power side channels and its value reflects whether there exists abnormal execution behavior.

We evaluate the proposed code execution tracking solution on a 8051 MCU, a popular choice for IoT, wearable devices, industrial sensors, etc, because of their ease of software development, royalty-free licensing and low cost, and small silicon footprint. We select nine programs as our benchmark suite. We demonstrate that our method can significantly improve the tracking accuracy. For the benchmark programs, we are able to achieve 99.94% accuracy in recovering the type of executed instruction, which is 42.55% higher than that of the previous method. Besides recovering instruction type, our method can identify which instruction in code is executed at a given moment, with the average accuracy of 98.56%. In addition, we demonstrate that our method is able to detect abnormal execution behavior effectively, even for the case when a single instruction in the original code has been changed.

The remainder of the paper is organized as follows. In Section 2, we present background knowledge and our problem formulation. Next, we give an overview of our method in Section 3, and we discuss our revised HMM and Viterbi algorithm in Section 4. Section 5 describes how to design observation symbol and emission distribution function. In Section 6, we explore how our method can facilitate detecting abnormal execution. We evaluate our method with STC89C52 MCU in Section 7 and discuss the limitations of our method in Section 8. At last, we introduce related works in Section 9 and conclude this work in Section 10.

2. BACKGROUND AND PROBLEM FORMULATION

In this section, we first discuss the importance of code execution tracking on key extraction attack and CFI. Then, we briefly describe CFG and HMM. Finally, we formulate the problem to be investigated in this work.

2.1 Key Extraction Attack

When cryptographic algorithms are implemented in software, for security reasons, designers usually choose implementations from open source libraries (e.g., OpenSSL [14]). Hence, adversaries are knowledgeable about the code, and typically extract keys via side-channel attacks or fault injection attacks. Both attacks require to precisely track code execution before launching the attacks.

Side-channel attacks analyze MCU's behavior on physical side-channels, e.g., acoustic emission [15] and power consumption [16], when vulnerable instruction pieces are executed during encryption or decryption. Such vulnerable instruction pieces are carefully chosen from the code, so that their operations are correlated to the key bits. For example, differential power analysis (DPA) [1] on Data Encryption Standard requires to obtain the power traces for the 16^{th} encryption round; correlation power analysis (CPA) [17] on Advanced Encryption Standard (AES) requires to obtain the power traces just after the first *AddRoundKey* operation.

Fault injection attacks attempt to disturb cipher's operations and extract keys by analyzing the cipher's faulty output [18, 19]. For each fault attack method, attackers must inject faults into MCU at precise timing when vulnerable instruction pieces are executed. Take extracting the round key of AES-128 encryption algorithm [20] as an example. The vulnerable sections can be the code between the 6^{th} and the 7^{th} *MixColumns* operations [21], in the last encryption round but before its *SubBytes* operation [2], in the previous rounds of the targeted round [22,23], and in the penultimate round but before its *MixColumns* operation [24].

Thus, all these key extraction attacks require to locate the vulnerable instruction pieces during execution. Consequently, code execution tracking is an essential step in such attacks and it is unfortunate that previous works in this domain often assume such method is readily available.

2.2 Control Flow Integrity

Code execution tracking is also the foundation for CFI techniques, which is effective to cope with control flow hijacking attacks, e.g., return oriented programming [25], jump oriented programming [26], buffer overflow [27] and firmware modification [3]. CFI tracks code execution and prevents any attempt to deviate execution flow from CFG [28].

Fine-grained CFI checking [7] requires adding a piece of CFI guard code before every control-flow instruction (e.g., indirect jump) and allocating an additional shadow stack to track and validate every function call, return, and exception during execution. To improve the performance, many CFI techniques require hardware support. For instance, ROPecker [8] relies on last Branch Recording (LBR), which is a hardware unit introduced in Intel's Nehalem architecture, for CFI, and hardware modification is needed when applying it to other processors. Similarly, HAFIX [10] depends on special hardware designs to track and verify function returns during execution, and other studies [11, 12] introduce dedicated hardware modules to track instruction execution sequence, calculate a signature for this sequence, and compare it with golden values.

Thus, the aforementioned CFI techniques require software or MCU modification and incur non-trivial overhead to the system. For embedded systems with limited resources, especially legacy embedded systems, such intrusive solutions are not applicable.

2.3 Basics for CFG and HMM

Control Flow Graph [29] is a directed graph and represents how a program can transit between basic blocks during execution. A basic block is a sequence of instructions

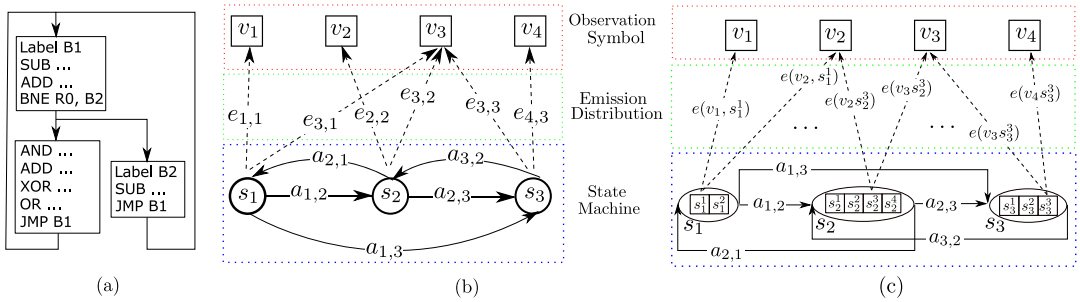

Figure 1: (a) An example Control Flow Graph. (b) Illustration of a classic HMM.(c) Illustration of a revised HMM.

that has only one entry point at the beginning and one exit point at the end. That is, a basic block can be considered as an execution primitive, and its instruction combination always run in the same order. Figure 1(a) shows an example of CFG: each node in CFG represents a basic block and each edge in CFG represents a valid control transfer between basic blocks.

A hidden Markov model [30] consists of three parts: state machine, emission distribution, and observation symbol. The visible observation depends on hidden states and the hidden state transition is a Markov process. Each state has a probability distribution over the possible observation. Therefore, the sequence of observation provides some information about the sequence of hidden states. Viterbi algorithm [30] is often used to find the most probable state sequence for a given observation sequence in HMM. Figure 1(b) shows an HMM example, which consists of three states ($\{s_1, s_2, s_3\}$) and four possible observation symbol values ($\{v_1, v_2, v_3, v_4\}$). At time t, people can make an observation o_t, where $o_t \in \{v_1, v_2, v_3, v_4\}$. By continuously observing the HMM, an observation sequence O is obtained. O is generated by the HMM going through a state sequence Q, where q_t ($q_t \in \{s_1, s_2, s_3\}$) in Q is the state of HMM at time t. If HMM is in state s_i at time t, it will jump to state s_j at time $t+1$ with probability $a_{i,j}$. The probability to observe v_k is $e_{k,i} = Pr[v_k|s_i]$, which depends on the hidden state s_i.

2.4 Problem Formulation

Term definition. For the sake of clarity, we define the following terms used throughout the paper before formulating the problem.

Instruction Instance. We use an *instruction instance* to indicate a specific instruction, including both its machine code and location in the code. If two instructions in the code have the same machine code but with different PC values, we treat them as different instruction instances. For the sake of simplicity, an *instruction sequence* in this paper refers to a sequence of instruction instances.

Instruction Type. Instruction type of an instruction is only determined by its operation code. We treat instruction instances with the same operation code belong to the same instruction type.

Formulation. Although both key extraction and CFI rely on code execution tracking, their requirements are different. For key extraction, they only need to accurately track the normal execution of the given code. In *normal execution*, the actual execution flow always obeys the CFG. On the contrary, control flow hijacking attacks usually introduce invalid control transfers [3, 25–27], and the actual execution flow deviates from the CFG, namely *abnormal execution*. The objective of CFI is therefore to detect whether there is abnormal execution in the system.

Thus, we formulate two sub-problems in this work.

1. **Normal Execution Tracking:** Given the source code and the power measurement traces during code execution, we would like to recognize which instruction instance is executed at each moment within the power traces.
2. **Abnormal Execution Tracking:** Given the source code and the power measurement traces during code execution, we would like to detect whether abnormal execution is performed.

3. OVERVIEW

In this section, we introduce the overall flow of our execution tracking method and discuss the main challenges. To simplify discussion, we consider every instruction costs one unit of time. Practically, some instructions may cost multiple units of time. In that case, we treat them as multiple one-unit instructions.

3.1 Overall Flow

While we formulated two problems to tackle in this work, both of them share the same code execution tracking framework. We model code execution on MCU and its power side-channel behavior as an HMM. To be specific, we consider a basic block as an individual state, control transfer between basic blocks as state transition, and the power consumption of MCU as observation. Then, tracking code execution is equivalent to recognizing how underlying state transition happens for a given power trace.

Workflow of Code Execution Tracking. The overall flow of our execution tracking framework is illustrated in Figure 2, which contains an HMM construction phase and an execution tracking phase. The final output of our framework includes two sequences: an instruction sequence and the corresponding likelihood of each instance in the sequence.

The HMM construction phase determines the parameter values of the HMM. We obtain substate (i.e., instruction instance), state and state transition information from the CFG of the given code, which can be derived by analyzing the disassembled binary [31]. Based on a set of power traces when executing various instructions, the observation symbols are obtained by performing signal extraction and

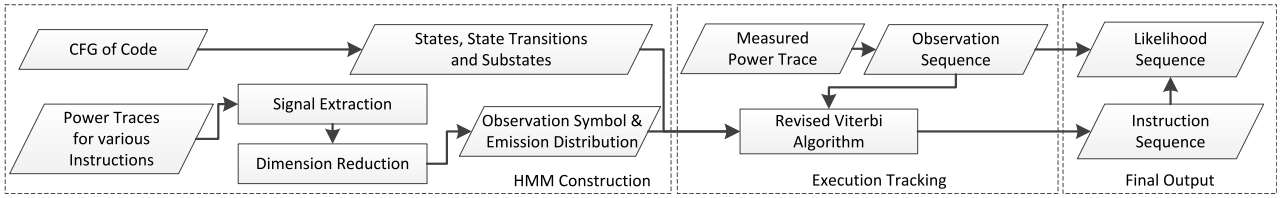

Figure 2: Workflow of the proposed code execution tracking framework.

dimension reduction, and emission distribution is modeled with Gaussian distribution, as detailed in Section 5.

In the execution tracking phase, we first obtain observation sequence from power trace, and then identify the most probable instruction sequence. In particular, we first divide power traces into chunks that map to individual instructions. This is a straightforward procedure because power trace exhibits periodical characteristics with periods mapping to instructions [13,32]. Then, to obtain the observation symbol value for each chunk, we conduct filtering and linear transformation on the raw power trace within the chunk, which correspond to signal extraction and dimension reduction, respectively. With our revised Viterbi algorithm, we can recover the most probable instruction sequence from the obtained observation sequence. Then, based on recovered instruction sequence and observation sequence, we can calculate the likelihood sequence.

Normal and Abnormal Execution Tracking. The reported instruction sequence directly addresses the normal execution tracking problem. To approach the abnormal execution tracking problem, we can examine the likelihood sequence, as detailed in Section 6.

3.2 Challenges

To track code execution with HMM, we could define individual instruction type or individual instruction instance as a state in HMM, but both have limitations. Using instruction type as state only recovers the instruction type sequence instead of instruction sequence, which cannot solve the normal execution tracking problem. Using instruction instance as individual state can solve this problem, but its computational complexity is prohibitive because the given code usually contains a large amount of instruction instances, which creates a large number of states. For instance, the space complexity of Viterbi algorithm is proportional to the number of states and hence becomes inefficient. In addition, every state requires an emission distribution function, and building individual emission distribution function for every instruction instance is impractical for large programs. To reduce the number of states without sacrificing recognition accuracy, we define a basic block in CFG as state in HMM. Because the instruction instances in a basic block always run in the same order, if we know how basic block transition occurs during execution, the instruction sequence can be determined as well.

The above state definition in HMM, however, incurs many challenges. Classic HMM defines emission distribution function on the entire state, and it needs to divide given power trace into chunks, where each chunk corresponds to one unknown state. However, basic blocks may contain various number of instruction instances and hence different states have unequal lengths in our case, which makes dividing power trace for unknown states non-trivial. Moreover, classic Viterbi algorithm assumes the length of every state is 1, and it cannot work for states with unequal lengths. To tackle these problems, we introduce substates, which represent instruction instances (see Figure 1(c)) in basic blocks, and define emission distribution function on substate. By doing so, we only need to divide power trace into chunks that correspond to instructions. We also revise Viterbi algorithm to work with unequal length states and the substates. By combining states and substates, we are able to simultaneously preserve the CFG information in HMM and dramatically reduce computational complexity.

To reduce the cost of building emission distribution function for every instruction instance, we build individual emission distribution function for each instruction type and instruction instances of the same type use the same distribution function. Such emission distribution function design suffices to track code execution, because different parts of the code usually have different instruction type sequences and accurately recognizing instruction type enables us to recover the underlying state. To reduce the noise in instruction type recognition, we try to extract high-quality signals from power traces and use them for observation symbols. To further reduce the computational overhead of distribution function construction, we also exploit dimension reduction when designing observation symbols.

4. REVISED HMM AND VITERBI ALGORITHM

In this section, we discuss how to revise HMM and Viterbi algorithm for the problems investigated in this work.

4.1 HMM Parameters

Figure 1(c) shows an example of our revised HMM, and formally, our HMM is characterized by the following parameters:

States, state lengths and substates. States are given by $S = \{s_i \mid 1 \leq i \leq N\}$, where N is the number of basic blocks in the CFG, and state s_i corresponds to the ith basic block in the CFG. We use l_i to represent the number of substates, i.e., instruction instances, in state s_i. Then s_i can be further represented as a sequence of l_i substates, i.e., $s_i = \{s_i^1, s_i^2, \ldots, s_i^{l_i}\}$, where s_i^m is the mth substate in s_i.

State transition probability and initial substate distribution. In our model, state transition represents control transfer between basic blocks. However, such transition probability distribution can vary significantly with different inputs to program, and the exact input for targeted execution is not available to us. Hence, we use $a(s_i, s_j)$ to indicate whether there is a valid control transfer in CFG from s_i to s_j, that is

$$a(s_i, s_j) = \begin{cases} 1 & \text{if transition is valid} \\ 0 & \text{otherwise} \end{cases} . \quad (1)$$

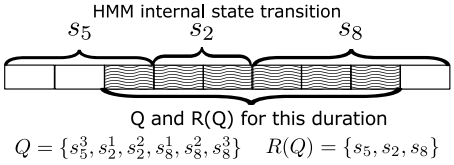

Figure 3: Example for Q and $R(Q)$.

Consequently, $a(s_i, s_j)$ is no longer a probability value, because, for any i, $\sum_{1\le j\le N} a(s_i, s_j)$ could be larger than 1. The first chunk in examined power trace can correspond to any instruction instance in any basic block. To obtain the prior probability of instruction sequence, we also need the probability of an instruction instance, i.e., substate, being the initial one. This probability distribution also varies with different input data, and we simply assume all substates have equal probability to be the initial one.

Observation symbols and emission distribution. We use V to represent the set of all possible observation symbol values. As mentioned above, our emission distribution is defined on substate. Emission distribution for substate s_i^m is given by $\{e(v, s_i^m) = p(v|s_i^m) \mid v \in V\}$. We detail how to design observation symbol and emission distribution function in Section 5.

4.2 Likelihood Estimation

Then, finding the most probable instruction sequence is equivalent to finding the most probable substate sequence. Next, we discuss how to estimate the probability for a substate sequence given the observation sequence.

When calculating the probability of a substate sequence Q, we also need its corresponding state sequence $R(Q)$. $R(Q)$ explicitly represents the state transitions in Q. For example, in Figure 3, for substate sequence $\{s_5^3, s_2^1, s_2^2, s_8^1, s_8^2, s_8^3\}$, its corresponding state sequence is $\{s_5, s_2, s_8\}$.

Formally, a substate sequence of length T can be written as $Q = \{q_1, q_2, \ldots, q_T\}$, where $q_t \in \bigcup_{1\le i\le N} s_i$. $R(Q)$ can be written as $R(Q) = \{r_1, r_2, ..., r_K\}$, where $r_k \in S$. Note that, $q_1 \in r_1$ and $q_T \in r_K$. We name r_1 as the initial state, and name r_K as the final state. Because q_1 and q_T can be any intermediate substate between r_1 and r_K, the corresponding substate sequence parts for r_1 and r_K in Q can be incomplete. Nevertheless, substate sequence parts for states between the initial state and the final state must be complete in Q.

Then, given an observation sequence $O = \{o_1, o_2, \ldots, o_T\}$ of length T, a candidate substate sequence Q and its corresponding state sequence $R(Q)$, the probability of Q given O is $p(Q|O) = p(Q, O)/p(O)$. Because $p(O)$ is the same for all candidate Qs, to find the most probable substate sequence, we only need to compare the $p(Q, O)$ part, given as,

$$\begin{aligned} p(Q, O) &= p(O|Q) \cdot p(Q) \\ &= \prod_{1\le t\le T} e(o_t, q_t) \cdot [b(q_1) \cdot \prod_{2\le k\le K} a(r_{k-1}, r_k)], \end{aligned}$$

where $b(q_1)$ is the probability that q_1 is the initial substate.

As indicated by Equation 1, $\prod_{2\le k\le K} a(r_{k-1}, r_k)$ equals 0 if $R(Q)$ contains any invalid state transition, otherwise it is $b(q_1)$, whose value is the same for different q_1. Consequently, the likelihood value J for a substate sequence Q given O can be estimated as,

$$J(Q, O) = \begin{cases} 0, & R(Q) \text{ contains} \\ & \text{invalid transition} \\ \prod\limits_{1\le t\le T} e(o_t, q_t), & \text{otherwise} \end{cases}. \quad (2)$$

Then, the most probable substate sequence is simply the one with the maximum J value.

4.3 Revised Viterbi Algorithm

Next, let us discuss how to efficiently find the most probable substate sequence, given observation sequence O of length T. Our algorithm follows the idea in classic Viterbi algorithm. We first calculate the J value of most probable substate sequence by recurrence, and then reconstruct the most probable substate sequence by backtracking.

For each state s_j and each time t, our revised Viterbi algorithm calculates a quantity, denoted by $\delta_t(j)$. When $1 \le t \le T$, $\delta_t(j)$ represents the maximal J for substate sequence that starts at time 1 and terminates at time t with $s_j^{l_j}$ (i.e., the last substate of s_j). So at time T, the calculated $\delta_T(j)$ corresponds to the maximal J value of substate sequence that ends exactly at the last substate of s_j. Given the observation can also end at any substate inside s_j (e.g., the case shown in Figure 3), we also calculate s_j's δ at time $T+1$ to $T+l_j-1$. When $T+1 \le t \le T+l_j-1$, $\delta_t(j)$ represents the maximal J for substate sequence which starts at time 1 and terminates at time T with $s_j^{l_j-(t-T)}$ as the last substate.

Let us first give the basic idea about how to calculate δ by recurrence. For a given substate sequence, we can divide it into two parts: one part corresponds to its final state and the other part is the substate sequence before its final state. For instance, we can divide the Q shown in Figure 3 into $\{s_8^1, s_8^2, s_8^3\}$ and $\{s_5^3, s_2^1, s_2^2\}$. Then, according to Equation 2, the J value of a valid substate sequence is the product of J value for its final state part and J value for the former part (i.e., the substate sequence part before the final state). It means, if a substate sequence terminates with $s_j^{l_j}$ at time $t(t > l_j)$ and its J value is $\delta_t(j)$, the J value of its former part must be one of the δ values at time $t - l_j$. Otherwise, there must be other substate sequence that also terminates at time t with $s_j^{l_j}$, has larger J than it. Consequently, we can calculate $\delta_t(j)$ based on δ values at time $t-l_j$, and all δ values can be obtained by recurrence. When recurrently calculating $\delta_t(j)$, we also use a quantity $\phi_t(j)$ to record which previous state's δ at time $t - l_j$ maximizes $\delta_t(j)$.

Next, we give the formal recurrence relation and initialization step for state s_j. For the sake of simplicity, we use $\Omega(s_j, m, n)$ to represent the partial substate sequence in state s_j, starting with s_j^m and terminating with s_j^n, i.e.,

$$\Omega(s_j, m, n) = \{s_j^m, s_j^{m+1}, \ldots, s_j^n\},$$

where $1 \le m \le n \le l_j$.

Recurrence. When $t \ge 1 + l_j$ and $t \le T$, according to Equation 2, we have

$$\begin{aligned} \delta_t(j) &= [\max_i \delta_{t-l_j}(i)] \cdot J(s_j, \{o_{t+1-l_j}, \ldots, o_t\}) \\ \phi_t(j) &= \operatorname*{argmax}_i \delta_{t-l_j}(i) \\ &\text{s.t. } a(s_i, s_j) = 1. \end{aligned} \quad (3)$$

When $t \geq 1+l_j$ and $T < t \leq T+l_j-1$, we tackle substate sequences terminating in the middle of state s_j at time T. In this case, δ is calculated by,

$$\delta_t(j) = [\max_i \delta_{t-l_j}(i)] \cdot J(s'_j, \{o_{t+1-l_j}, \ldots, o_T\})$$
$$\phi_t(j) = \operatorname{argmax}_i \delta_{t-l_j}(i)$$
$$\text{s.t. } a(s_i, s_j) = 1,$$

where $s'_j = \Omega(s_j, 1, T-t+l_j)$.

Initialization. $\delta_t(j)$ and $\phi_t(j)$ for $1 \leq t \leq l_j$ are set by initialization. Because l_j may be greater than T, we have

- when $1 \leq t \leq l_j$ and $1 < t \leq T$

  $$\delta_t(j) = J(s'_j, \{o_1, \ldots, o_t\}), \quad \phi_t(j) = 0,$$

  where $s'_j = \Omega(s_j, l_j - t + 1, l_j)$.
- when $1 \leq t \leq l_j$ and $t > T$

  $$\delta_t(j) = J(s'_j, \{o_1, \ldots, o_T\}), \quad \phi_t(j) = 0,$$

  where $s'_j = \Omega(s_j, l_j - t + 1, l_j - t + T)$.

$\phi_t(j) = 0$ indicates s_j, terminating at t, is the initial state.

With above, if a substate sequence is of length T and uses s_j as final state, its maximal J value is given by

$$max\{\delta_T(j), \ldots, \delta_{T+l_j-1}(j)\}.$$

Hence, the J value of the most probable substate sequence is given by

$$\max_j\{\max\{\delta_T(j), \ldots, \delta_{T+l_j-1}(j)\}\}.$$

Once the J value of the most probable substate sequence is located, we can reconstruct the most probable substate sequence by backtracking the ϕ value accordingly, similar to classic Viterbi algorithm.

4.4 Complexity Analysis

In this subsection, we analyze our method's complexity, by comparing it to the naive method that treats each instruction instance as individual state and uses classic Viterbi algorithm to solve it. Because the instructions inside a basic block always run in the same order, it means most instruction instances in the code only have single possible previous instruction. However, classic Viterbi algorithm updates the δ value for a state at time t by enumerating all states' δ values at time $t-1$ and records the ϕ value for every state at every moment, which is unnecessary for most instructions.

Suppose a program has X instruction instances and Y basic blocks. Because we usually observe MCU execution for a long time, the observation sequence length, denoted by T, should be much larger than the length of the longest basic block, denoted by l_{max}. Both classic Viterbi algorithm and our revised one can be implemented in a dynamic programming manner.

The space complexity is mainly determined by the size of the array used in dynamic programming, which records ϕ and δ for every state at every moment. Then, the space complexity of the naive solution is $O(T \times X)$, and ours is $O((T + l_{max} - 1) \times Y) = O(T \times Y)$. Let us take `aes` case shown in Table 1 as an example. l_{max} in `aes` is 82. If T is 7065, then the size of the array with the naive method is $7065 \times 1472 \approx 10^7$ and that with our method is $(7065 + 82 - 1) \times 55 \approx 3.9 \times 10^5$. Hence, we can reduce the memory overhead by about 96.1%.

The time complexity is mainly determined by the cost of updating elements in the array. In both methods, the cost of updating one element consists of two parts: one is evaluating the likelihood of the state given the observation, e.g., $J(s_j, \{o_{t+1-l_j}, \ldots, o_t\})$ in Equation 3, and the other one is enumerating previous states, e.g., $\max_i \delta_{t-l_j}(i)$ in Equation 3. Assume the complexity of calculating the likelihood for one instruction instance is $O(1)$. In our method, let us consider the worst case that every element is updated with Equation 3. Then at time t, Y states need to evaluate the likelihood for X instructions in total and each state needs to enumerate Y previous states. Hence, the total time complexity is $O(T \times (X + Y^2))$. With the naive method, there are X states. At time t, it also needs to evaluate X instructions, but each state needs to enumerate X states. Therefore, the total time complexity is $O(T \times (X + X^2))$, which is much larger than ours.

5. OBSERVATION SYMBOL AND EMISSION DISTRIBUTION FUNCTION

A good observation symbol design should enable us to recover the instruction sequence accurately, and reduce the overhead of building emission distribution function at the same time. In order to achieve the above objectives, we need to solve the following two problems.

First, because we build individual emission distribution function for each instruction type, we should design the observation symbol in such a manner that it facilitates recognizing instruction type. Consequently, signal extraction techniques are employed to increase the correlation between observation symbol and instruction type.

Second, because a chunk of power trace that corresponds to one instruction instance could contain hundreds of sample points, there is significant overhead to model the distribution of such a high-dimensional variable. Therefore, dimension reduction technique is used to reduce computational complexity.

In the following, we first discuss our signal extraction technique, and then present the overall design flow of our observation symbol.

5.1 Signal Extraction

From the viewpoint of frequency domain, power signal is synthesized from different frequency components. The objective of signal extraction in this work is to select those frequency components that are highly correlated to instruction type and filter out other components.

Frequency Components Selection. The raw power signal represents the total power consumption of the MCU, and there are at least four factors that affect MCU power consumption when an instruction is executed. First, instruction type affects power consumption by designating the micro operations of the processor. Next, when executing an instruction, the instruction operands and the instruction executed prior to it affect the low-level switching activities of the circuit. Finally, environment noise would also have some impact on the obtained power trace. The last three factors would interfere with instruction type recognition, and their impact can be mitigated by increasing the correlation between observation symbol and instruction type.

Usually, a frequency component can be represented by its amplitude value A_{com}. As raw power signal is determined by four factors, we simply model A_{com} as linear combination of two parts, given by

$$A_{com} = A_{type} + A_{other}, \quad (4)$$

where A_{type} is determined by instruction type, and A_{other} represents the part determined by instruction operand, previous executed instruction and environmental noise together.

We assume A_{type} and A_{other} in Equation 4 are independent. Ideally, different instruction types have different A_{type} values and the same type of instruction has the same A_{type} value. In this case, to evaluate the correlation between a frequency component and instruction type, we can use the correlation between A_{type} and A_{com} instead.

We use *Pearson's correlation coefficient* to evaluate the correlation. Then the correlation between A_{type} and A_{com} is given by

$$\rho(A_{com}, A_{type}) = \frac{cov(A_{com}, A_{type})}{\sqrt{D_{com} D_{type}}},$$

where $cov(A_{com}, A_{type})$ is the covariance between A_{type} and A_{com}, D_{com} is the variance of A_{com}, and D_{type} is the variance of A_{type}.

Because A_{type} and A_{other} are independent, we have

$$\rho(A_{com}, A_{type}) = \frac{cov(A_{type}, A_{type}) + cov(A_{other}, A_{type})}{\sqrt{D_{com} D_{type}}} = \frac{D_{type} + 0}{\sqrt{D_{com} D_{type}}} = \sqrt{\frac{D_{type}}{D_{com}}}.$$

Therefore, we should select those frequency components with the following two characteristics,

1. D_{type}/D_{com} should be as large as possible in order to obtain larger correlation $\rho(A_{com}, A_{type})$.

2. In addition, the magnitude of D_{type} should be as large as possible. Larger D_{type} means the difference on A_{type} among different instruction types is more significant, which is easier to be captured.

Evaluating D_{com} and D_{type}. First, evaluating D_{com} is simple, because A_{com} value can be obtained by transforming the raw power signal from time domain to frequency domain. Second, although we cannot measure A_{type} directly, D_{type} can be evaluated as follows. Because A_{other} and A_{type} are independent, if A_{other} keeps constant during sampling, the conditional variance of A_{com} in this case is equal to D_{type}, according to Equation 4. As a result, to evaluate D_{type}, we can sample A_{com} by randomly changing instruction types while keeping instruction operand and previous executed instruction fixed. To make environmental noise constant, we can measure the power trace for every instruction instance multiple times and use the averaged power trace instead. To further improve the accuracy, we can calculate D_{type} multiple times with different configurations of instruction operands and previous executed instruction, and use the averaged value when comparing different components.

Filtering. Once the appropriate frequency components are selected, it is straightforward to obtain the filtered power signal. That is, we can obtain the frequency amplitude spectrum of one power trace with Fast Fourier Transformation, zero out the amplitude values of those inappropriate frequency components, and generate the filtered power trace by Inverse Fast Fourier Transformation.

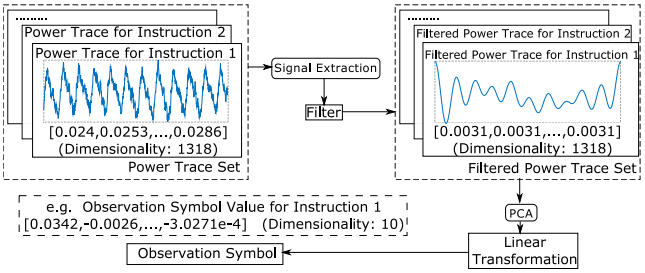

Figure 4: Observation Symbol Design Flow.

5.2 Overall Design Flow

Figure 4 shows our observation symbol design flow. The input is a set of power traces with various instruction instances. Among these instruction instances, the instruction type, the instruction operand and instruction executed prior to the sampled instruction are all randomly changed.

First, we conduct signal extraction according to the given set of power traces and generate the filtered power traces. Next, we conduct dimension reduction with principle component analysis (PCA) [33]. PCA can generate a linear transformation function that maps high-dimensional power signal to a lower-dimensional signal, while the transformation preserves useful information as much as possible. When applying PCA, we need to decide the dimensionality of the obtained lower-dimensional signal. To solve this problem, we evaluate how dimensionality affects instruction type recognition rate with statistical classifiers, e.g., Naive Bayes classifier, and use the smallest dimensionality contributing to the highest recognition rate.

Finally, we use the low-dimensional signal obtained after applying PCA as our observation symbol. For each instruction type, we fit its emission distribution with Multivariate Gaussian Distribution Model, based on the above power trace set.

6. ABNORMAL EXECUTION TRACKING

Till now, we have shown how to recover the instruction sequence, and solve the normal execution tracking problem. In this section, we discuss how to detect abnormal execution, based on the fact that in abnormal execution cases, the most probable sequences typically have a reduced likelihood compared to the normal execution cases. Then, we discuss the possibility that attackers can evade our tracking method.

6.1 Detection via Likelihood Sequence

When invalid control transfers are introduced by control flow hijacking attack, our revised Viterbi algorithm would recognize the actual instruction sequence, deviating from CFG, as another valid instruction sequence that obeys CFG and has the largest probability to generate the observation sequence. Because the actual instruction sequence, containing abnormal execution, intends to implement a malicious function that does not exist in the original CFG, the actual instruction sequence's instruction type sequence is usually different from that of any valid instruction sequence defined by the CFG. Therefore, the type sequence of the actual

instruction sequence is different from that of the reported instruction sequence. With the above, when tracking abnormal execution, some instruction instances in the actual sequence would be incorrectly recognized as the wrong type of instruction instances in the reported sequence. This phenomenon can thus be used for abnormal execution tracking.

Ideal Case. Ideally, when tracking normal execution, instruction instances should be all correctly recognized. Hence, if we could distinguish between correctly recognized instruction instances and incorrect ones, we are able to detect abnormal execution. In order to achieve this objective, we examine the likelihood of the reported instruction instance m given the corresponding observation v, i.e., $e(v, m)$, which is also the conditional probability of v given m. When m is reported with incorrect recognition, the corresponding observation, denoted by v_{inc}, is actually generated by another instruction of different type. Because different instruction types usually generate different observations, m is not likely to generate v_{inc}. If m is reported with correct recognition, the corresponding observation, denoted by v_c is generated by m itself. Hence, we have $e(v_{inc}, m) < e(v_c, m)$.

Calibrated Likelihood. Motivated by the above, for each instruction instance in the code, we record its average likelihood value in normal execution. When detecting abnormal execution, for each instruction instance in the reported instruction sequence, we subtract the recorded average likelihood for this instruction instance from its current likelihood value, and we name the obtained difference as **calibrated likelihood**. Then, the calibrated likelihood sequence is given by

$$\{e(o_1, q_1) - h(q_1), e(o_2, q_2) - h(q_2), \ldots, e(o_T, q_T) - h(q_T)\},$$

where $h(q_t)$ is the average likelihood value of the instruction instance q_t in normal execution.

If the instruction instance is correctly recognized, its calibrated likelihood should be around zero. Otherwise, the calibrated likelihood for incorrectly recognized instruction instance should be biased to be negative.

Note that, although an additional average likelihood number is recorded for each instruction instance, this overhead is much smaller than that of building and recording individual emission distribution function for each instruction instance.

6.2 Security Analysis

Given different instruction types may have the same power emission model, an attacker may try to launch a mimic attack, which evades our detection by constructing adversarial instruction sequence whose power consumption fingerprint happens to be valid. In this section, we discuss the probability of such attack.

Threat Model. To construct a malicious sequence to fulfill an adversary's hidden agenda from scratch is challenging. We imagine that an adversary will utilize the well-known attack (i.e., Call-Preceded Return-Oriented- Programming, in short CPROP [25]) to reuse the existing code to accomplish this goal. Thus, for illustration purpose, we analyze the likelihood of mimic attackers that utilize CPROP. CPROP maliciously redirects the target of the `ret` instruction to a wrong instruction whose preceding instruction is a `call` instruction. Without loss of generality, suppose all control transfers in a program are caused by function calls or returns. Then CPROP constructs adversarial instruction sequence by introducing invalid transitions among basic blocks. We assume attackers know the power fingerprint of every basic block in the code.

We assume that the target program has m basic blocks, each basic block has n instructions (excluding the final control transfer instruction), and each basic block has v ($v \leq m$) valid next basic blocks in CFG (i.e., outdegree of any node in CFG is v). Given different instruction types may have the same power emission, we assume that the instruction set contains a instruction types in total and can be divided into b groups, where each group contains a/b instruction types on average and the instruction types in the same group have the same power emission. We can only distinguish instruction types from different groups via power side-channel.

When CPROP wants to insert a malicious basic block after a legitimate basic block, she creates an invalid transition. To evade detection, CPROP should create the malicious basic block so that its power fingerprint is the same as one of the original v valid ones. In the worst case, CPROP can use one of the $m-v$ basic blocks (i.e., the ones that create invalid transition) as the malicious one and the resulting adversarial instruction sequence only deviates from the valid instruction sequence by a single basic block. Let us denote the probability that such adversarial sequence evades detection by P_{evade}. If the instructions in basic blocks are randomly and independently distributed,

$$P_{evade} = 1 - [1 - (\frac{1}{b})^n]^{v(m-v)}. \quad (5)$$

The second term in Equation 5 gives the probability that any of the $m - v$ malicious candidates has a different power fingerprint from those of v valid ones, i.e., the probability of detecting the adversarial sequence. We can expand this term in Binomial series, then

$$P_{evade} = kx - \sum_{t=2}^{k}[(-1)^t \frac{k(k-1)\ldots(k-t+1)x^t}{t!}], \quad (6)$$

where $x = (\frac{1}{b})^n$ and $k = v(m-v)$.

When $kx < 1$, the absolute value of $(-1)^t \frac{k(k-1)\ldots(k-t+1)x^t}{t!}$ decreases as t increases. Hence, the second term on the right hand side of Equation 6 is always positive. Then we have

$$P_{evade} < kx = \frac{v(m-v)}{b^n}.$$

For the code size m and the basic block size n that are typical for mid-size embedded devices, the magnitude of kx is small and P_{evade} is close to 0 with the following reasons. First, b^n is exponentially proportional to n. Second, different instruction types' power emissions usually provide sufficient diversity and b is large. For example, $b \approx 10^2$ for the MCU used in our experiments, because $a = 152$ and the accuracy of classifying instruction types can be estimated by b/a, which is 70% in our case as shown in Section 7.2.3. Third, $k \leq 0.25m^2$, $kx \leq m^2/4b^n$. Hence, kx is small for a typical code size m and a basic block size n. For instance, for a target code of 10^6 basic blocks, we only need $n > 7$ to guarantee $P_{detect} > 99.75\%$ with the MCU used in our experiments.

Thus, the probability of constructing an adversarial instruction sequence with a valid power consumption fingerprint is close to 0 in general.

7. EVALUATION

In this section, we conduct various experiments to evaluate the proposed solution. First, we describe the hardware and software platforms used for evaluation and introduce the performance metrics used in this work. The evaluation results are divided into four parts: designing observation symbol, tracking normal execution, tracking abnormal execution, and tracking execution on different chips.

7.1 Experimental Setup

MCU under test. The method proposed in this paper actually can be applied to any MCU model, as long as the execution time of every instruction is a constant. Many MCU architectures in current market satisfy this requirement, such as PIC12 [13] , 8bit AVR [32] and Intel's 8051 [34]. STC89C52, an implementation of 8051 architecture, is used in this evaluation. Since there is no external RAM on its evaluation board, instructions relevant to external RAM (e.g., `MOVX`) are excluded from evaluation. Most instructions in this MCU cost only one machine cycle. For instructions costing 2 or 4 machine cycles, we treat them as 2 or 4 different single-cycle instructions. As a result, the effective instruction set contains 152 different single-cycle instruction types. This MCU is clocked at $11.0592MHz$ using an external oscillator.

Power measurement. To measure the power consumption of the MCU under test, a resistor of 46.7Ω is placed between VCC pin of the MCU and its power supply, and the voltage drop over it is measured using a Tektronix MDO3034 oscilloscope with sampling rate of $1.25GS/s$.

Benchmark programs. Our benchmark suite consists of 9 programs, in which eight of them are from Dalton Project [35] that is used to evaluate the performance of 8051 MCUs. The remaining one is an implementation of AES-128 encryption algorithm migrated to our MCU. The details of these nine programs are shown in Table 1, including the number of instruction instances (# of Inst.), the number of basic blocks (# of BB.), and the length of instruction sequence tracked during program execution (Measured Inst.). For the programs `matrix`, `aes`, `pid` and `dct`, we only measure 7065 executed instructions, because their power traces for a complete execution will go beyond the maximal length that can be measured with our experimental setup. For all the other programs, power traces of a complete execution are recorded.

Evaluation Metrics. We evaluate our method with two metrics. The first one is *Instruction Sequence Accuracy (ISA)*, which demonstrate the accuracy of the recognized instruction instance in the reported instruction sequence. The second metric is *Type Sequence Accuracy (TSA)*, which only measures the accuracy of the recognized instruction types. TSA is a more important metric, because the performance of some configurations (to be introduced below) cannot be measured by ISA and our method relies on the instruction type information to track program execution.

We compare our work with the method proposed in [13], which recovers the instruction type sequence by treating every instruction type as a state in classic HMM, and instruction type transition probabilities are extracted from code under test. Their observation symbol is obtained by conducting dimension reduction on raw power signal directly.

Name	Description	# of Inst.	# of BB.	Measured Inst.
aes	AES-128	1427	55	7065
sqroot	Square root	1002	98	3800
sort	Bubble sort	233	37	4430
matrix	Matrix multiplication	413	30	7065
pid	Simulate cruise control in car	1572	199	7065
dct	Discrete cosine transform	560	51	7065
gcd	Euclidean algorithm	69	11	135
fib	Fibonacci sequence	159	24	782
csumex	Cumulative sum chart	89	12	665

Table 1: Benchmark suite.

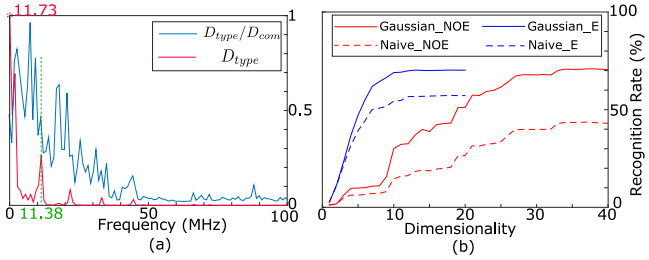

Figure 5: (a) Normalized D_{type}/D_{com} and D_{type} for different frequency components. (b) Classifying instruction type after PCA, when signal extraction is used (E) and not used (NOE).

We denote the HMM defined in [13] with prefix `TYPE` and our proposed HMM with prefix `BB` (means Basic Block). We use suffix `E` and `NOE` to indicate whether signal *extraction* technique is used or not in designing observation symbol. Therefore, there are four configurations to be evaluated: TYPE_NOE, TYPE_E, BB_NOE and BB_E, where TYPE_NOE corresponds to the method in [13]. All the four configurations run on the same server with Intel Xeon E5-2609 CPU and 16GB RAM.

7.2 Observation Symbol Design

In this section, we demonstrate how to design observation symbol and its impact on instruction type recognition accuracy. In our experiment, the set of power traces used for designing observation symbol consists of about 180,000 power traces from various instruction instances, measured on the same chip.

7.2.1 Signal Extraction

Let us first estimate D_{type}/D_{com} and D_{type}. We obtain the frequency amplitude spectrum of the power traces with Fast Fourier Transofrmation. Figure 5(a) shows D_{type}/D_{com} and D_{type} within frequency range $0 \sim 100MHz$, where the value of D_{type}/D_{com} and D_{type} are normalized for presentation. Based on our discussion in Section 5.1, we only select frequency components within range $(0MHz, 11.38MHz)$, because both D_{type}/D_{com} and D_{type} within this range are larger than those outside of this range and they are used for instruction type recognition.

Configuration	TSA(%)									Average
	aes	csumex	dct	fib	gcd	matrix	pid	sort	sqroot	
TYPE_NOE	48.09	99.52	73.26	56.59	81.18	86.69	37.04	92.20	56.42	70.11
TYPE_E	57.72	99.61	70.77	93.48	78.53	95.62	56.57	97.89	57.62	78.65
BB_NOE	99.89	100.00	100.00	100.00	98.24	99.97	93.43	100.00	99.75	99.03
BB_E	99.88	100.00	100.00	100.00	100.00	99.98	99.80	100.00	99.78	99.94
	ISA(%)									
BB_NOE	90.55	100.00	100.00	100.00	98.24	99.97	92.81	100.00	97.04	97.62
BB_E	90.49	100.00	100.00	100.00	100.00	99.98	99.48	100.00	97.09	98.56

Table 2: TSA and ISA in normal execution tracking. The last column shows the average value for each row.

7.2.2 Dimension Reduction with PCA

To decide the dimensionality of the final low-dimensional signal, we examine the instruction type recognition rate with Naive Bayes classifier and Gaussian Bayes classifier.

Figure 5(b) shows the instruction type recognition rates for different classifiers after applying PCA, when signal extraction is used and not used, respectively. For both classifiers that we have tested, cases with signal extraction require only about 10 dimensions to achieve the maximum recognition rate, meanwhile the non-filtered cases require much more dimensions (about 35 shown in the figure) to achieve the same value. Given this, if signal extraction is used, we use signals consisting of the first 10 dimensions after applying PCA as observation symbol, otherwise we use signal consisting of first 35 dimensions after applying PCA as observation symbol. For both cases, emission distribution function of each instruction type is built with Multivariate Gaussian Model.

7.2.3 Effectiveness of Signal Extraction

We have another two observations from Figure 5(b). First, as fewer dimensions are required when signal extraction is used, it means our signal extraction technique can reduce the complexity in building the emission distribution function with Multivariate Gaussian Model. Second, with Naive Bayes classifier, the maximal recognition rate for case with signal extraction is larger than that of case without signal extraction. This means, by selecting frequency components of larger D_{type}/D and D_{type}, we can recognize instruction types more accurately. But the improvement on the maximum recognition rate almost disappears for the Gaussian Bayes classifier case, where the maximum recognition rate is about 70% for both cases. One possible reason is that, Gaussian Bayes classifier considers the dependency among different dimensions that can help instruction type recognition, and the improvement introduced by signal extraction is thus much smaller.

7.3 Normal Execution Tracking

Table 2 lists the accuracy of normal execution tracking for different configurations with different programs. For each configuration and each program, we track its execution for five times and the average accuracy value is reported in the table. From table 2, we have the following observations.

First, using BB model can always achieve higher TSA than using TYPE model. No matter whether signal extraction technique is used or not, with BB model, the TSA for tracking all 9 programs is over 93%. In particular, when BB_E configuration is used, the TSA is always over 99.7%. For configurations with TYPE model, TSA varies a lot and is quite low in some cases. For example, TYPE_NOE only achieves 37.04% TSA for `pid`, while it achieves 99.52% TSA for `csumex`. On average, our most powerful method BB_E can outperform TYPE_NOE by 42.55%. This is consistent with our expectation, because BB model preserves more knowledge from CFG and hence it has a higher probability to track the execution correctly.

Second, on average, signal extraction technique can improve TSA by 12.18% for TYPE model, and 0.92% for BB model. Earlier we observed that the maximum recognition rate with Gaussian classifier almost keeps unchanged, no matter whether signal extraction is used or not. This is not contradictory to our observation here. The difference lies in experimental setup, i.e., in instruction type classification experiment, the instruction type is uniformly distributed, which is different from the distributions in actual programs.

Third, configurations with BB model can also achieve very high ISA, which is over 97% on average. It demonstrates that, by precisely recovering the executed instruction's type, it is sufficient for our method to track which instruction instance in the code is executed. On some programs, ISA is lower than TSA, such as `aes`. We manually check the results of `aes` case, and find there are two basic blocks in the program which only differ at one location in their instruction type sequences. At this location, one block uses `XRL A,R0` instruction and the other one uses `XRL A,direct` instruction. These two instructions both implement exclusive-or function, differ in addressing mode, and belong to different types. When one of them is incorrectly recognized as the other one, TSA only treats this `XRL` instruction is incorrectly recognized while ISA regards all the instructions in the basic block are incorrectly recognized. Therefore, ISA is much smaller than TSA in this case.

To sum up, our BB model outperforms the original TYPE model significantly. The signal extraction technique further improves execution tracking accuracy.

7.4 Abnormal Execution Tracking

Because designing a full-fledged CFI method is beyond the scope of this work, in this subsection, we mainly demonstrate that abnormal execution could decrease the reported calibrated likelihood values, compared to that of normal execution cases.

We use firmware modification attack as an example. Intuitively, less modification on the original code is more difficult to be detected. Given this, we first study single instruction replacement, insertion and deletion cases on `aes` program, which do not change the control transfers after modification. In these three cases, we respectively replace one `NOP` instruction with an `ADD A,0x00` instruction, insert a new `NOP` instruction, and delete an existing `NOP` instruction. All these modifications are conducted at the beginning of

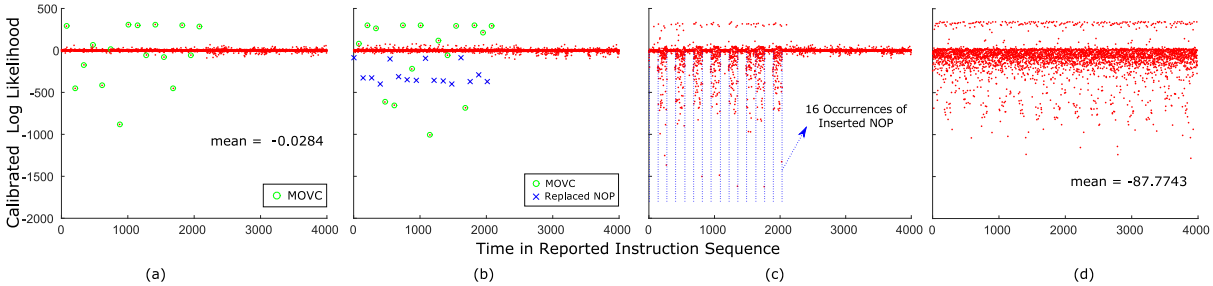

Figure 6: Calibrated log likelihood of the first 4000 instruction instances in reported instruction sequence for (a) normal execution, (b) single instruction replacement, (c) single instruction insertion, and (d) multi-instruction modification.

SubByte function in `aes` program, which is called 16 times within one measurement. Next, to study multi-instruction modification case, we simulate an attack that replaces `aes` with `dct` during execution. Each power measurement covers 7065 instructions and BB_E configuration is used as execution tracking method.

Figure 6 shows the calibrated log likelihood sequences in the attack cases and the normal execution case. Because the magnitude of the original likelihood value is sometimes quite small, we perform calibration on log likelihood instead. From the results, we have the following observations.

For the normal execution case (Figure 6(a)), the calibrated likelihood for most instruction instances are close to zero and the mean value is -0.0284. Although several calibrated likelihood values in the sequence deviate from zero a lot, indicated by green circle, all these values correspond to the same instruction instance whose type is `MOVC A,@A+DPTR`, and their mean value is -43. Most registers in 8051 are 8-bit registers, but DPTR consists of 16 bits. Hence, the power consumption of `MOVC` is more sensitive to operands than other instructions, and its calibrated likelihood varies more significantly.

For the replacement case (Figure 6(b)), the distribution of the calibrated likelihood is significantly biased with a large negative mean value. Based on the reported instruction sequence, we can group the multiple occurrences of the same instruction instance and observe its calibrated likelihood's distribution. The `ADD A,0x00` instruction after replacement is incorrectly recognized as `NOP`. All sixteen occurrences of the replaced `NOP`, indicated by blue cross in the figure, have negative calibrated likelihood value, and their mean is -287. This replaced `NOP` can be easily distinguished from the above-mentioned `MOVC` instruction. The calibrated likelihood values of `MOVC` instruction instance can be both negative and positive, indicated by green circle, and their mean value is -56.

For the insertion and deletion cases, the observations are similar and the result for the insertion case is given in Figure 6(c) as an example. We observe that the calibrated likelihood for many instruction instances around the inserted (or deleted) `NOP`, indicated by blue dash line, is biased to a large negative value. This is because, instruction insertion/deletion can cause multiple instruction instances around it to be incorrectly recognized. For example, if we delete the first instruction from a 4-instruction basic block, the second instruction in this basic block can be incorrectly considered as the start of this state during tracking, and the remaining three instructions in this basic block together with one instruction from the next basic block in the actual execution flow may be incorrectly recognized as one state, which affects the following basic block.

For the multi-instruction modification case (Figure 6(d)), the calibrated likelihood for most instruction instances is biased to be negative, and the mean value is -87.7743, which is much smaller than that in the normal execution case. The degradation of the calibrated likelihood here is more significant than the single instruction modification cases, which is consistent with the intuition that multi-instruction modification is easier to be detected.

7.5 Execution Tracking on Different Chips

Chip No.	TSA(%)			
	TYPE_NOE	TYPE_E	BB_NOE	BB_E
Chip1	44.84	79.04	93.66	99.94
Chip2	75.28	79.57	99.62	99.93
Chip3	67.94	79.51	99.60	99.93
Chip4	73.26	71.50	99.62	99.92
Avg.	65.33	77.40	98.13	99.93
STD	14.007	3.943	2.976	0.007

Table 3: Average TSA for four configurations on different chips. The last two rows show the average and standard deviation for each column.

In this subsection, we demonstrate that emission distribution function built with power traces from one chip (e.g., Chip0) can be used to track code executions on other chips (e.g., Chips 1∼4) from the same architecture family. Results for normal execution tracking on Chip0 are listed in Table 2, and the TSA results on Chip1∼4 using the same emission distribution model derived from Chip0 are shown in Table 3. These data can lead us to the following observations.

First, the emission distribution model derived from Chip0 work very well on Chips 1∼4. When comparing the average TSA accuracy of all chips (e.g., the Avg. row in table 3 and the last column of table 2), we found that they are very close to each other, even though applying the model to different chips can still introduce small accuracy loss (the largest TSA degradation is 6.82% with TYPE_NOE, and the degradation for BB_E is almost zero). There are two possible reasons for such similarity. First, the power consumption of each instruction is largely determined by its instruction types, bacause instructions of the same type share many on-chip hardware modules that contribute most of the overall power consumption, as shown in Figure 5. Second, different chips used in this experiment have the same architecture and similar layouts, so power consumptions of each instruction type are very close among different chips. Although they may still have some unique features, the variances introduced by such differences are small.

Another observation is that the TSA values for configurations with signal extraction techniques are more stable. This is shown by the standard deviation (i.e., the STD row) in Table 3 where BB_E and TYPE_E have much smaller standard deviation than BB_NOE and TYPE_NOE does. This is because signal extraction facilitates to eliminate certain frequency components that are more sensitive to the difference between multiple chips (e.g., the static power corresponding to frequency 0).

8. LIMITATIONS AND FUTURE WORK

Though our method has significantly reduced the computational complexity compared to the naive solution, it can still induce undesired overhead when the target program is large and contains a large number of instructions and basic blocks. Such a situation can get exacerbated when interrupts are enabled during execution, because interrupts can be triggered at any time during code execution and create various valid control transfers from every instruction instance to the beginning of interrupt service routines. Under such circumstances, every instruction instance in the code becomes one individual basic block and it results in a larger number of states. To tackle this problem, a hierarchical code execution tracking method can be used. For instance, when an interrupt is triggered, a processor needs to perform special operations, e.g., context switching. It is possible to first identify such operations from power traces, determine the power traces corresponding to the execution of interrupt service routines, and remove them. Then, we can concatenate the remaining power trace segments, and conduct code execution tracking on the newly-constructed power trace.

In our experiments, the execution of the benchmark programs do not use peripheral devices, and hence the measured power trace is mainly contributed by the MCU itself. When peripheral devices are used, however, the correlation between instruction type and measured power trace would decrease and it may result in reduced accuracy in code execution tracking. As a direction for our future work, one can increase the measuring points of power traces. That is, instead of measuring the overall power consumption of the system, we would collect power traces from multiple power pins on the MCU and investigate their correlations with the executed instructions, thereby mitigating the impact of the peripheral devices on proposed code execution tracking method.

9. RELATED WORKS

In this section, we briefly discuss related works, including execution tracking via digital channels, normal execution tracking with side-channels, abnormality detection, and code reverse engineering.

Execution Tracking via Digital Channel. Some ARM-based MCUs (e.g., Cortex-M3) contain a dedicated hardware unit for code execution tracking, namely embedded trace macrocell (ETM) [36]. However, it is usually not practical to cycle-accurately track code execution at normal CPU speed with ETM. Moreover, many MCUs do not have such hardware support for execution tracking.

Normal Execution Tracking via Side-channel. Eisenbarth *et al.* [13] utilized HMM to recover the instruction type sequence during code execution. It treats an instruction type as a state, and extracts transition probabilities between instruction types. However, solely recovering instruction type is not able to locate instruction instance. Msgna *et al.* [32] tried to track execution flow by modeling one basic block in CFG as a state with classic HMM. However, their method cannot tackle the general case where basic blocks have unequal length. Compared to the above works, our code execution tracking method can locate the exact instruction instance during execution accurately.

Abnormality Detection via Side-Channel. Some works detect abnormal execution by calculating the cross correlation between examined execution's side-channel trace, e.g., power trace [37, 38] and RF trace [39, 40], and the corresponding side-channel trace of golden execution. In practice, however, it is difficult to determine the exact golden execution flow because embedded system's execution usually interacts with changeable environment and varies a lot in different runs. By contrast, the detection technique based on our tracking method has no such requirement. WattsUpDoc [41] uses statistical tools to classify every 5-second power trace chunk's corresponding execution to be normal or abnormal, where features, such as mean and variance, are used for classification. We have shown calibrated likelihood is a good feature for abnormal execution detection, and it can be used to enhance WattsUpDoc.

Side-channel Based Code Reverse Engineering. These methods focus on recovering the code in the system, instead of tracking the execution flow for a given code. Vermon *et al.* [42] recovered the bytecodes running on a Java smart card. However, this method requires calculating the average power trace of the targeted sequence of bytecodes, which is impractical for the general cases. Novak [43] and Clavier [44] showed how to recover the substitution tables of secret A3/A8 algorithm, but their method is limited to recovering the look-up table part. Goldack and Paar [45] proposed to recover the type of single instruction instance by building power consumption templates for every instruction type. However, their template models the distribution of raw power signal after simple dimension reduction. We have demonstrated that dimension reduction itself does not lead to high recognition accuracy, but it can be improved by our signal extraction technique.

10. CONCLUSION

This paper proposes a non-intrusive yet highly-accurate code execution tracking method for embedded systems utilizing power-side channel. This is achieved with signal extraction scheme to improve instruction type recognition and a revised Viterbi algorithm for effective instruction sequence extraction. Experimental results show that our method is able to track code execution accurately in normal execution tracking and effectively capture code modification in abnormal execution tracking.

11. ACKNOWLEDGMENTS

This work was supported in part by the Chinese University of Hong Kong internal grant No. 4055049, Hong Kong S.A.R. Research Grants Council (RGC) under Early Career Scheme No. 24207815, in part by National Natural Science Foundation of China (NSFC) under Grant No. 61432017, 61532017, 61572415, and 61472358, and in part by National Science Foundation (CNS-1513107).

12. REFERENCES

[1] P. C. Kocher, *et al.* Differential power analysis. In *Proc. of Advances in Cryptology (CRYPTO)*, 1999.

[2] P. Dusart, *et al.* Differential fault analysis on A.E.S. In *Proc. of Applied Cryptography and Network Security (ACNS)*, 2003.

[3] A. Cui, *et al.* When firmware modifications attack: A case study of embedded exploitation. In *NDSS*, 2013.

[4] A. Francillon and C. Castelluccia. Code injection attacks on harvard-architecture devices. In *Proc. of Conference on Computer and Communications Security (CCS)*, 2008.

[5] T. Goodspeed. Exploiting wireless sensor networks over 802.15. 4. In *Texas Instruments Developper Conference*, 2008.

[6] M. Abadi, *et al.* Control-flow integrity. In *Proc. of Conference on Computer and Communications Security (CCS)*, 2005.

[7] Ú. Erlingsson, *et al.* XFI: software guards for system address spaces. In *Proc.s of Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] Y. Cheng, *et al.* Ropecker: A generic and practical approach for defending against ROP attacks. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2014.

[9] V. Pappas, *et al.* Transparent ROP exploit mitigation using indirect branch tracing. In *Proc. of USENIX Security Symposium (USENIX Security)*, 2013.

[10] L. Davi, *et al.* HAFIX: hardware-assisted flow integrity extension. In *Proc. of Design Automation Conference (DAC)*, 2015.

[11] M. Milenkovic, *et al.* Hardware support for code integrity in embedded processors. In *Proc. of International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, 2005.

[12] F. A. T. Abad, *et al.* On-chip control flow integrity check for real time embedded systems. In *Proc. of Cyber-Physical Systems, Networks, and Applications (CPSNA)*, 2013.

[13] T. Eisenbarth, *et al.* Building a side channel based disassembler. *Transactions on Computational Science*, 2010.

[14] OpenSSL. https://www.openssl.org/.

[15] D. Genkin, *et al.* RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Proc. of Advances in Cryptology (CRYPTO)*, 2014.

[16] N. Benhadjyoussef, *et al.* The research of correlation power analysis on a aes implementations. *Journal of Intelligent Computing Volume*, 2011.

[17] E. Brier, *et al.* Correlation power analysis with a leakage model. In *Proc. of Cryptographic Hardware and Embedded Systems (CHES)*, 2004.

[18] J. Balasch, *et al.* An in-depth and black-box characterization of the effects of clock glitches on 8-bit mcus. In *Proc. of Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2011.

[19] A. Dehbaoui, *et al.* Electromagnetic transient faults injection on a hardware and a software implementations of AES. In *Proc. of Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2012.

[20] NIST FIPS Pub. Advanced encryption standard (AES). *Federal Information Processing Standards Publication*, 2001.

[21] P. Derbez, *et al.* Meet-in-the-middle and impossible differential fault analysis on AES. In *Proc. of Cryptographic Hardware and Embedded Systems (CHES)*, 2011.

[22] Y. Liu, *et al.* DERA: yet another differential fault attack on cryptographic devices based on error rate analysis. In *Proc. of Design Automation Conference (DAC)*, 2015.

[23] R. Lashermes, *et al.* A DFA on AES based on the entropy of error distributions. In *Proc. of Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2012.

[24] A. Moradi, *et al.* A generalized method of differential fault attack against AES cryptosystem. In *Proc. of Cryptographic Hardware and Embedded Systems (CHES)*, 2006.

[25] N. Carlini and D. Wagner. ROP is still dangerous: breaking modern defenses. In *Proc. of USENIX Security Symposium (USENIX Security)*, 2014.

[26] T. K. Bletsch, *et al.* Jump-oriented programming: a new class of code-reuse attack. In *Proc. of Symposium on Information, Computer and Communications Security (ASIACCS)*, 2011.

[27] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 1996.

[28] N. Carlini, *et al.* Control-flow bending: On the effectiveness of control-flow integrity. In *Proc. of USENIX Security Symposium (USENIX Security)*, 2015.

[29] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, 1970.

[30] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 1989.

[31] C. Zhang, *et al.* Practical control flow integrity and randomization for binary executables. In *Proc. of Symposium on Security and Privacy (SP)*, 2013.

[32] M. Msgna, *et al.* The b-side of side channel leakage: Control flow security in embedded systems. In *Proc. of Security and Privacy in Communication Networks (ICST)*, 2013.

[33] I. Jolliffe. *Principal component analysis*. 2002.

[34] I. S. MacKenzie. *The 8051 microcontroller*. 1998.

[35] UCR Dalton Project. http://www.cs.ucr.edu/~dalton/.

[36] Embedded Trace Macrocells. http://www.arm.com/products/system-ip/debug-trace/trace-macrocells-etm/.

[37] C. R. A. González and J. H. Reed. Detecting unauthorized software execution in sdr using power fingerprinting. In *MILITARY COMMUNICATIONS CONFERENCE, 2010-MILCOM 2010*, 2010.

[38] C. R. A. Gonzalez and J. H. Reed. Power fingerprinting in sdr integrity assessment for security and regulatory compliance. *Analog Integrated Circuits and Signal Processing*, 2011.

[39] S. Stone and M. Temple. Radio-frequency-based anomaly detection for programmable logic controllers in the critical infrastructure. *International Journal of Critical Infrastructure Protection*, 2012.

[40] S. J. Stone, *et al.* Detecting anomalous programmable logic controller behavior using rf-based hilbert transform features and a correlation-based verification process. *International Journal of Critical Infrastructure Protection*, 2015.

[41] S. S. Clark, *et al.* Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices. In *2013 USENIX Workshop on Health Information Technologies, HealthTech '13*, 2013.

[42] D. Vermoen, *et al.* Reverse engineering java card applets using power analysis. In *Proc. of Information Security Theory and Practices (WISTP)*, 2007.

[43] R. Novak. Side-channel attack on substitution blocks. In *Proc. of Applied Cryptography and Network Security (ACNS)*, 2003.

[44] C. Clavier. Side channel analysis for reverse engineering (SCARE) - an improved attack against a secret A3/A8 GSM algorithm. *IACR Cryptology ePrint Archive*, 2004.

[45] M. Goldack and I. C. Paar. Side-channel based reverse engineering for microcontrollers. *Master's thesis, Ruhr-Universität Bochum, Germany*, 2008.