Hiding in Plain Sight? On the Efficacy of Power Side Channel-Based Control Flow Monitoring

Yi Han, Matthew Chan, and Zahra Aref, *Rutgers University;* Nils Ole Tippenhauer, *CISPA Helmholtz Center for Information Security;* Saman Zonouz, *Georgia Tech*

https://www.usenix.org/conference/usenixsecurity22/presentation/han

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Hiding in Plain Sight? On the Efficacy of Power Side Channel-Based Control Flow Monitoring

Yi Han
Rutgers University

Matthew Chan
Rutgers University

Zahra Aref
Rutgers University

Nils Ole Tippenhauer
CISPA Helmholtz Center for Information Security

Saman Zonouz
Georgia Tech

Abstract

Physical side-channel monitoring leverages the physical phenomena produced by a microcontroller (e.g. power consumption or electromagnetic radiation) to monitor program execution for malicious behavior. As such, it offers a promising intrusion detection solution for resource-constrained embedded systems, which are incompatible with conventional security measures. This method is especially relevant in safety and security-critical embedded systems such as in industrial control systems. Side-channel monitoring poses unique challenges for would-be attackers, such as: (1) limiting attack vectors by being physically isolated from the monitored system, (2) monitoring immutable physical side channels with uninterpretable data-driven models, and (3) being specifically trained for the architectures and programs on which they are applied to. As a result, physical side-channel monitors are conventionally believed to provide a high level of security.

In this paper, we propose a novel attack to illustrate that, despite the many barriers to attack that side-channel monitoring systems create, they are still vulnerable to adversarial attacks. We present a method for crafting functional malware such that, when injected into a side-channel-monitored system, the detector is not triggered. Our experiments reveal that this attack is robust across detector models and hardware implementations. We evaluate our attack on the popular ARM microcontroller platform on several representative programs, demonstrating the feasibility of such an attack and highlighting the need for further research into side-channel monitors.

1 Introduction

Detection of malicious code execution on a platform is challenging in general [2], in particular for embedded computer systems. The embedded setting imposes a host of limitations, such as constrained computational power and a lack of hardware support for security features [43]. In many cases, embedded systems also have strict real-time operating deadlines; importantly, this is true of many safety-critical and industrial control system (ICS) applications. Previous malicious code execution attacks on these systems (e.g. Stuxnet [14], BlackEnergy [31], and many others [23]) have been carried out by nation-state-level adversaries, costing untold amounts in damage to critical infrastructure and service downtime.

One promising defense for embedded systems against these attacks is physical side-channel monitoring [3, 22, 32, 39]. It measures physical phenomena such as transient power consumption or electromagnetic radiation (both a result of circuit transistor switching) in order to monitor for and accurately detect malicious behavior.

Physical side-channel monitors pose unique challenges for would-be adversaries. Unlike traditional monitoring in the software realm, physical side-channel signals are a consequence of code being executed on a chip. As a result, physical signals are difficult or impossible to be spoofed or manipulated by an attacker [4]. Side-channel monitors also limit potential attack vectors by their air-gapped nature, i.e. they cannot be interacted with except through the microcontroller-emitted signals that they monitor.

These obstacles are thought to make conventional attacks ineffective, and promote the idea that physical side-channel monitors provide a high level of security. Despite those claims, these systems have not yet been thoroughly evaluated due to the complexity of doing so. Recent work on evaluating control flow integrity defenses [29] has shown that the afforded practical security rarely matches up with the claimed theoretical security. Likewise, side-channel monitors may be resting on the assumption that the barriers to attack that they raise are unlikely to be scaled (i.e., difficulties created by side-channel monitors, as mentioned above, prevent adversaries from performing attacks on the protected system efficiently or on a large scale).

In this work, we address the question: "Are physical side-channel monitors as secure as commonly thought (given that they can observe all attacker code executed)"—or in other words: *can an attacker hide in plain sight?* We present a novel attack against physical side-channel monitoring systems which demonstrates that they are vulnerable to an adaptive adversary who knows that a side-channel monitor is in use. Our attack uses a carefully-crafted assembly-level malware injection to produce a side-channel signal that can evade detection by a side-channel monitor. We exploit vulnerabilities of the data-driven models used in side-channel monitoring, finding adversarial programs which behave maliciously but evade detection. To facilitate reproduction of our approach and results (in contrast to prior work), we have made our code open source[1].

We summarize our main contributions as follows:

- We present a novel attack highlighting the design-level vulnerabilities of physical side-channel monitors to adversarial examples.
- We present a methodology for crafting such adversarial attacks on side-channel monitors, discussing how we approach the related challenges to create stealthy and functional malware.
- We evaluate our attack using the popular ARM Cortex-M processor on various control programs and control attacks. We show that our attack approach can find an evading sample in all cases.

2 Background

Physical Side-Channels. Physical side-channels refer to physical phenomena produced as a side-effect of system operations in digital circuits. Specifically, the execution of instructions as well as data read and write cause CMOS components in the digital circuits to switch on and off. This creates varying currents and voltages. Such varying current and voltage values can be observed by looking at the voltage fluctuations in the power consumption in the circuits. Such voltage fluctuations, called power side-channel signals, can be captured by measuring the voltage at the VCC pins of a digital chip (e.g., a micro-processor). Executing different instructions or transferring different data values across a data bus create different power side-channel signal patterns. The signal patterns are also affected by noise in the circuits. Nevertheless, power signals can be used to infer system execution within a chip. There are other physical side-channels such as electromagnetic (EM), acoustic, and thermal side-channels, however in this paper we focus mainly on power side-channel signals.

Physical Side-Channel Monitors. Physical side-channel monitors have been employed both in academia [22, 32, 39] and industry [3] to monitor the security of a system. The main advantage of physical side-channel monitors compared to traditional software-based monitors is that they are air-gapped, meaning that they are implemented externally to the monitored system. This provides isolation and a separate attack surface, reducing the number of available attack vectors.

[1]https://github.com/yihan0512/HidingInPlainSight.git

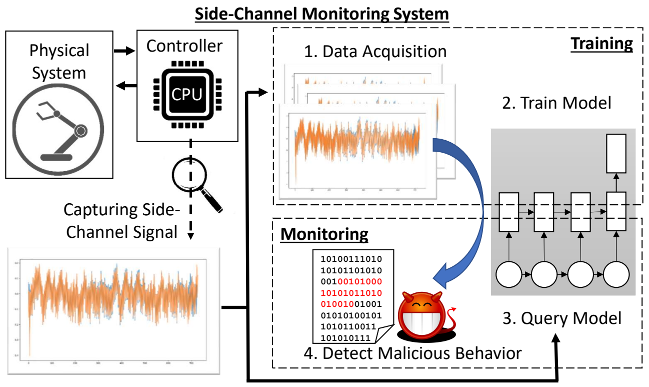


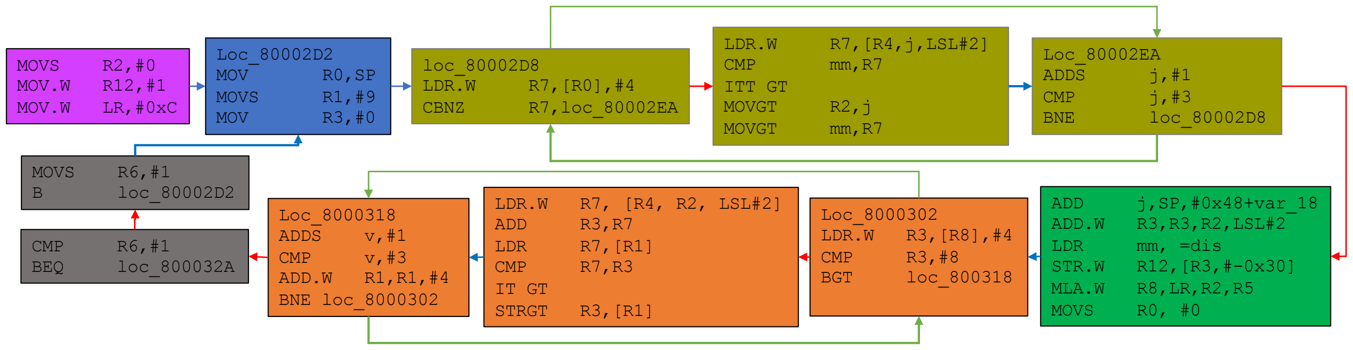
Figure 1: A side-channel monitor, consisting of (1) data acquisition, (2) model training, and (3, 4) querying the model to monitor behavior.

Physical side-channel monitors monitor the execution of a program though physical side-channel signals. A physical side-channel monitor consists of two parts, a physical side-channel collection module and an anomaly detector. The signal collection module collects physical side-channel signals during the execution of the program. The anomaly detector predicts the status of the program (e.g., normal or abnormal, malicious or benign) based on the collected signals.

The setup of a physical side-channel monitor is illustrated in Figure 1. They are commonly trained using data-driven models since current embedded platforms are exceedingly diverse and complex, discouraging the use of manually designed models. In the signal collection module, physical side-channel signals are collected during the normal execution of the program. This allows for initial training in a clean environment as well as subsequent retraining when necessary over time. The collected signals represent a wide subset of possible behaviors and control flows of the program. Once a model has been trained to sufficiently high accuracy, the monitoring phase begins. During system operation, the monitor measures the system, returning a response on whether the system is executing benign or malicious code.

Formalizing Side-Channel Restrictions. Pierazzi et al. [44] provide a formalized understanding of the challenges that a side-channel domain poses to an adversary. They differentiate between the *problem space* – the attacker-modifiable program code – and the *feature space*, which in this case is the resulting side-channel signal and input to the monitor. Without knowing an invertible or differentiable mapping between the problem and the feature space, standard gradient-based adversarial attacks [19] are infeasible.

Although the estimation or derivation of such a mapping poses an interesting question, we note that discovering such a mapping is outside the scope of this work, as any such mapping would be to some extent platform specific and therefore not widely generalizable. Additionally,



```
MOVS     R2,#0
MOV.W    R12,#1
MOV.W    LR,#0xC
```

```
Loc_80002D2
MOV          R0,SP
MOVS         R1,#9
MOV          R3,#0
```

```
loc_80002D8
LDR.W       R7,[R0],#4
CBNZ        R7,loc_80002EA
```

```
LDR.W       R7,[R4,j,LSL#2]
CMP         mm,R7
ITT GT
MOVGT       R2,j
MOVGT       mm,R7
```

```
Loc_80002EA
ADDS          j,#1
CMP           j,#3
BNE           loc_80002D8
```

```
MOVS     R6,#1
B        loc_80002D2
```

```
CMP      R6,#1
BEQ      loc_800032A
```

```
Loc_8000318
ADDS     v,#1
CMP      v,#3
ADD.W    R1,R1,#4
BNE loc_8000302
```

```
LDR.W    R7, [R4, R2, LSL#2]
ADD      R3,R7
LDR      R7,[R1]
CMP      R7,R3
IT GT
STRGT    R3,[R1]
```

```
Loc_8000302
LDR.W    R3,[R8],#4
CMP      R3,#8
BGT      loc_800318
```

```
ADD      j,SP,#0x48+var_18
ADD.W    R3,R3,R2,LSL#2
LDR      mm, =dis
STR.W    R12,[R3,#-0x30]
MLA.W    R8,LR,R2,R5
MOVS     R0, #0
```

Figure 2: Assembly code snippet of the path planning program. Different colors are used to show the mappings between instructions and corresponding power side-channel signal segments in Figure 3 in a clearer way.

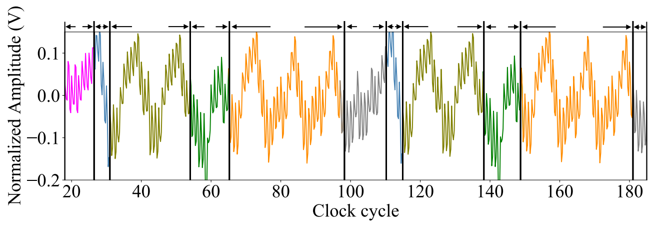


Figure 3: Power side-channel signal collected during the execution of the benign program. Colors correspond to the basic blocks of the program in Figure 2.

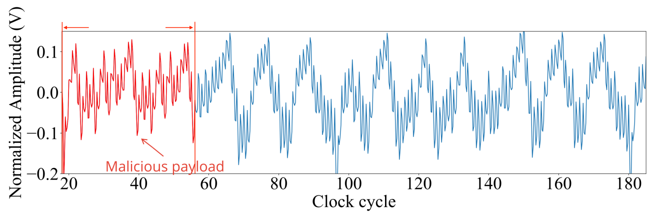


Figure 4: Power side-channel signal after the malicious payload is injected at the beginning. The malicious signal segment is marked in red.

in our testing we found that the resulting physical signal at each point in time was affected by both the currently executing instruction as well as the series of previously executed instructions, indicating that deriving a mapping would be non-straightforward. Instead, we utilize a problem-driven search [11, 45] to learn effective attack strategies. We also focus on several of the major challenge categories identified by Pierazzi et al. [44], which highlight the domain-specific adversarial barriers which make designing an attack on side-channel monitors difficult: problem-space transformation limitations and semantic preservation. The program code is based on a set of assembly instructions which have a rigid structure (feasible control flows), as well as semantic constraints such as temporal and data dependencies. The resulting discrete set of possible modifications must be taken into consideration by an attacker to preserve program functionality during an attack and remain evasive.

3 Problem Formulation

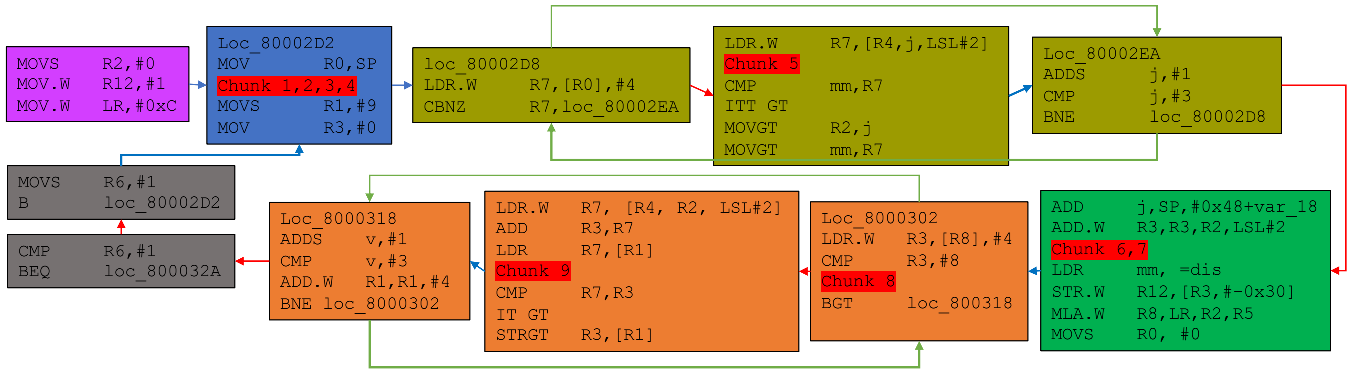
3.1 Motivating Example

Consider an embedded system executing a path-planning task for a robotic arm. This system is protected by a power side-channel based control flow monitor similar to those presented in related work and described in the previous section. Figure 2 and Figure 3 show a highly simplified example of a typical cyclical control program and corresponding power signal collected by the monitor. The signal represents an execution trace of the path-planning program: which instructions are executed, which branches are taken, and which control flow is followed. The anomaly detector monitors these power side-channel signals and reports any anomalous behavior.

More specifically, the anomaly detector takes the power side-channel of a scan cycle as input and outputs a confidence score. The confidence score indicates how likely it is that the input signal corresponds to an execution of the benign program. A threshold is set on the confidence score, with a confidence score lower than the threshold indicating an attack on the embedded system. The detector itself uses a data-driven model trained with power side-channel signals collected over a period of time during normal system operation.

An adversary wants to perform a data injection attack on the robotic arm, e.g., altering the inputs to the path planning algorithm. In this way, they can alter the internal state of the program and hence the output actuation to cause undesired arm behavior or damage.

The adversary also wants to launch a stealthy attack, i.e., the attack should not cause unintentionally observable effects. For example, replacing the original program with a malware program entirely might trigger an alarm in the system supervisor or other automated tools, as a the original data-monitoring feed is no longer available. Stealthy attacks have a more lasting impact on the system compared to attacks which break down the system quickly, as shown in real world attack examples, e.g., Stuxnet [24]. To achieve this, the adversary might choose to inject a malicious payload into the benign program.



```
MOVS     R2,#0
MOV.W    R12,#1
MOV.W    LR,#0xC
```

```
Loc_80002D2
MOV          R0,SP
Chunk 1,2,3,4
MOVS         R1,#9
MOV          R3,#0
```

```
loc_80002D8
LDR.W        R7,[R0],#4
CBNZ         R7,loc_80002EA
```

```
LDR.W        R7,[R4,j,LSL#2]
Chunk 5
CMP          mm,R7
ITT GT
MOVGT        R2,j
MOVGT        mm,R7
```

```
Loc_80002EA
ADDS         j,#1
CMP          j,#3
BNE          loc_80002D8
```

```
MOVS      R6,#1
B         loc_80002D2
```

```
CMP       R6,#1
BEQ       loc_800032A
```

```
Loc_8000318
ADDS      v,#1
CMP       v,#3
ADD.W     R1,R1,#4
BNE loc_8000302
```

```
LDR.W     R7, [R4, R2, LSL#2]
ADD       R3,R7
LDR       R7,[R1]
Chunk 9
CMP       R7,R3
IT GT
STRGT     R3,[R1]
```

```
Loc_8000302
LDR.W     R3,[R8],#4
CMP       R3,#8
Chunk 8
BGT       loc_800318
```

```
ADD       j,SP,#0x48+var_18
ADD.W     R3,R3,R2,LSL#2
Chunk 6,7
LDR       mm, =dis
STR.W     R12,[R3,#-0x30]
MLA.W     R8,LR,R2,R5
MOVS      R0, #0
```

Figure 5: Assembly code snippet of the path planning program with injected malware chunks. Code is otherwise unchanged from Figure 2.

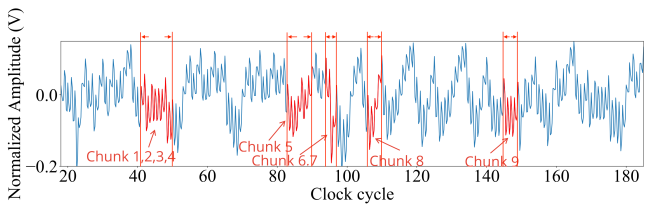


Figure 6: Power side-channel signal after the malicious payload is injected based on our proposed attack approach. The malicious signal segments are marked as red.

The signal in Figure 4 is from a version of the program where the malicious payload is injected at the beginning. This signal is not recognized by the side-channel monitor, and the resulting confidence score is lower than the threshold. Therefore, the anomaly detector rejects it and triggers an alarm.

Using the attack approach proposed in this paper, an adversary can find the optimal strategy to inject the malware into a legitimate binary (see Figure 5) while keeping the confidence score high. Figure 6 shows the corresponding power side-channel signal.

3.2 Research Goals and Challenges

This paper has the following goal: given a benign program guarded by a side-channel monitor and a malicious payload, inject the payload into the program without being detected by the monitor. As we consider direct attacks on the monitor itself to be out of scope (e.g., as it is air-gapped), the attacker has to ensure that the side-channel signals lead to a misclassification by the monitor. In particular, an adversary must craft an adversarial signal indirectly by modifying the program execution. The adversary also wants to make sure the desired malicious functionality remains intact. This gives rise to the main challenges that we address in this paper:

1. Side-channel signals are a consequence of code execution, meaning that malware cannot directly interact with the side-channel monitor and rendering gradient-based approaches ineffective. We address this challenge in Section 4 with a side-channel-aware malware optimization using an iterative search approach.
2. Side-channel monitors are trained on specific platforms for specific programs, using uninterpretable (and inaccessible to the adversary) data-driven models. We address this issue with the use of a substitute setup (Section 4.1), which we show can sufficiently replicate the original monitor for designing attacks.
3. Crafting an adversarial example is subject to domain-specific constraints, such as temporal (e.g. B happens after A) and data dependencies. To handle these constraints, we propose leveraging dependency analysis and chunking heuristics in Section 4.2.
4. Attacks need to be optimized for robustness to reliably evade detection even under the influence of measurement noise. We minimize the effect of noise by incorporating measurement variation in the optimization (Section 4.3) to produce a high-confidence, low-variance attack.

These challenges result in a highly constrained problem that highlights the strengths of side-channel monitors, but also their weaknesses against advanced adversaries.

3.3 System Model

We consider safety-critical embedded systems monitored by a power side-channel monitor. Examples of such systems include optimal path planning in robotic arms, traffic collision avoidance control for an aircraft, and common control algorithms such as the PID controller and Kalman filter. These programs are commonly run on an embedded controller such as a Programmable Logic Controller (PLC), which performs several tasks. The PLC reads physical measurements from sensors, runs a cycle of the control program, and sends the outputs to system actuators. Additionally, the PLC is connected to a Supervisory Control and Data Acquisition (SCADA) system which handles data logging and PLC programming.

Externally monitoring this system in real time is done using a power side-channel monitor. It continuously collects power side-channel signals of the embedded controller, and sends the signals to an anomaly detector. The anomaly detector inspects the side-channel signal for malicious behavior. It is trained using signals from executions of the embedded controller in a non-compromised environment and outputs a confidence score, with a high score (above a threshold) representing benign program execution and low score (below a threshold) indicating abnormal operation.

3.4 Attacker Model

The goal of the attacker is to compromise system operation by injecting a maliciously modified program while remaining undetected by the classifier. These adversarial evasion attacks can serve several purposes which we later evaluate, such as false data injection or confidential information disclosure.

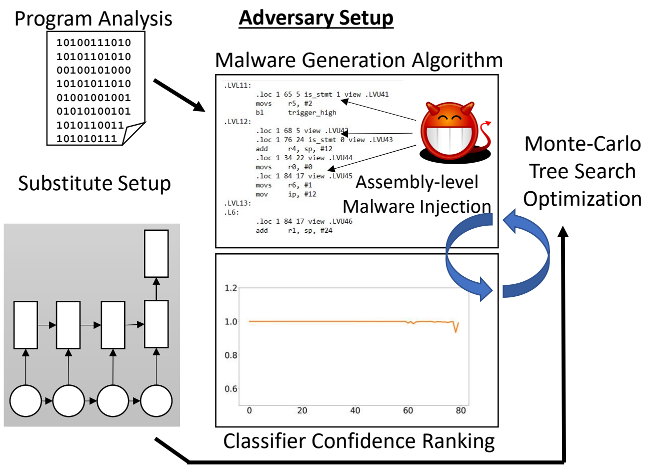
Due to the air-gapped nature of side-channel monitors, attacks directly targeting the monitor or its feature space (sampled points of the physical signal) are out of the scope of this work, and the attacker is limited to modifying the program running on the embedded system in the problem space (executed code). We assume that the attacker has knowledge of the hardware platform, the capability to upload a malicious program (e.g. having compromised the SCADA connected to the embedded controller), and has access to the original program running on the embedded system. This capability and attack vector is similar to Stuxnet [24].

Otherwise, we assume a black-box model with respect to the data-driven side-channel monitor. In the black-box case, the attacker is usually limited to external querying of the model to gain information about it; however, in the side-channel context, doing so is impractical as querying a live model would result in being detected. Instead, the attacker can leverage knowledge of the system's software and hardware to train a substitute setup [41], optimizing an attack and then transferring it back to the original model.

4 Attack Design

An overview of our attack approach is shown in Figure 7. The goal of our attack is, given a benign program and a malicious payload, finding a way to inject the malicious payload into the benign program such that the malicious code is undetected by the side-channel detector, i.e., an evading sample. Accomplishing this requires addressing two baseline considerations.
Syntactic Correctness. Fundamentally, the resulting program should be free of any syntax errors and able to be compiled. This is achieved as our adversary has sufficient information about both the program and malware to test and verify the feasibility of the resulting code.

Moreover, during execution the injected program should not crash. For example, this can happen if the malicious code unintentionally modifies a register that the benign code is currently using. Using program analysis and execution context-saving (analogous to multi-threading), we ensure that the



```
.LVL11:
        .loc 1 65 5 is_stmt 1 view .LVU41
        movs    r5, #2
        bl      trigger_high
.LVL12:
        .loc 1 68 5 view .LVU42
        .loc 1 76 24 is_stmt 0 view .LVU43
        add     r4, sp, #12
        .loc 1 34 22 view .LVU44
        movs    r0, #0
        .loc 1 84 17 view .LVU45
        movs    r6, #1
        mov     ip, #12
.LVL13:
.L6:
        .loc 1 84 17 view .LVU46
        add     r1, sp, #24
```

Figure 7: Using a substitute setup, an adversary can train attacks on a copy of the side-channel monitor.

malicious code does not accidentally overwrite registers the benign program uses.
Semantic Preservation. Additionally, the malicious functionality of the malware should be preserved after its injection into the benign program. For instance, if a malware intends to modify program inputs, it must do so before the point where they are used to have the "correct" effect. This creates locality requirements on where the malware can be placed. To address this, we perform program analysis on the benign program to identify a set of feasible locations where sections or "chunks" of the malware can be injected without impacting semantics.

Furthermore, as shown in past work [17], a typical malware size with specific adversarial goal in the context of embedded controllers (e.g., the controller's output value corruption) is often small (e.g., around 2KB [17]). That said, chunking the malware up and distributing its effects across various points of the side-channel signal is still necessary in practice to evade the detector.

Next, we detail how these considerations are addressed in our attack construction, starting with the physical setup.

4.1 Building a Substitute Setup

The isolation of the side-channel detector along with its real-time nature prevents an adversary from manipulating the side-channel detector freely, as described in Section 3. Therefore, we take the step of creating a substitute setup to help us find an evading sample. Specifically, the adversary procures a copy of the program running in the target system. This can be done in several ways, such as through a compromised SCADA, insider or physical system access, or even through the public availability of commonly-used controller software online. The adversary runs symbolic execution on the program to obtain test inputs that correspond to all execution paths of the program. The adversary then prepares a system with the same micro-controller model and peripheral circuits and uploads the program to this system. They then send the test inputs generated above to the system while at the same time collecting power side-channel signals from the system in the same way as the original system. Finally, a substitute side-channel detector can be trained using the collected signals. A side-channel based detector takes a power side-channel signal x as input, outputs a confidence score s

$$s = f(x), \quad s \in [0,1]. \tag{1}$$

A high confidence score indicates a normal execution while a low score indicates an anomalous/malicious one. Depending on the knowledge the adversary has, they can construct the detector with the same model architecture as the original setup or determine a "best guess" model architecture. We show in Section 5 that it is possible to find an evading sample even with a different model architecture.

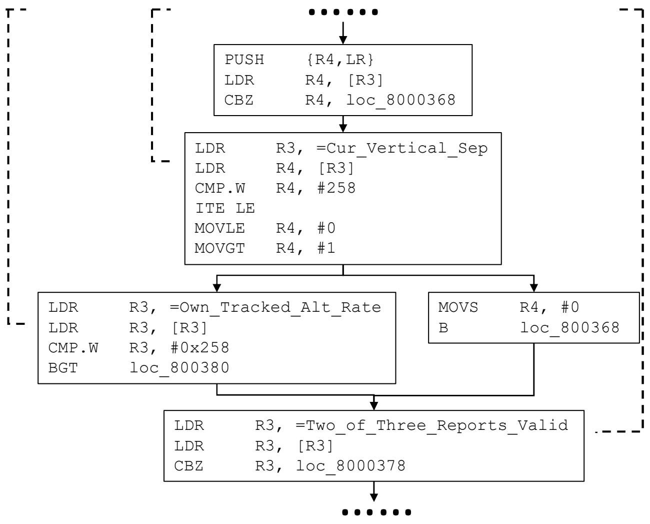
4.2 Malicious Payload Injection

In this subsection we describe our approach for injecting the malicious payload into a benign program. To allow for further analysis, we first disassemble the binary of the benign program. We then split the malicious payload into chunks. Next, we apply program analysis to the assembly code to identify where a malicious chunk can be injected for each chunk. Then, based on the optimization algorithm described in Section 4.3, the malicious chunks are injected into the benign program. Finally the injected program is assembled and uploaded to the device. We describe the key steps below.

Malicious Chunks. We propose splitting a malicious payload into chunks such that we can distribute them across the benign program. In this way, it is more likely that we can find an evading sample with higher confidence scores. We split the code such that each chunk is context-independent; in other words, each chunk can be injected and executed alone. For example, in the case of a data injection attack, instructions that correspond to the modification of each input variable can be considered as an individual chunk. We describe how we define the malicious chunks for all other attacks we consider in this paper in Section 5.

The Live Range of a Malicious Chunk. To ensure semantic preservation of the malicious payload, i.e., to make sure the malicious functionality is preserved, the locations that a malicious chunk can be injected are limited. We use the concept of a "live range" to express this constraint. We define the live range of a malicious chunk to be the set of locations in the benign program where a malicious chunk would still be effective when injected.

The live range is contextually dependent on the goals of the attack and results of the program analysis. Figure 8 represents the live ranges of several input variables in the scenario of a false data injection attack. Each attacked input variable needs to be modified after it is defined but before it is used. Otherwise, the intended effect of the input modification



```
PUSH    {R4,LR}
LDR     R4, [R3]
CBZ     R4, loc_8000368
```

```
LDR     R3, =Cur_Vertical_Sep
LDR     R4, [R3]
CMP.W   R4, #258
ITE LE
MOVLE   R4, #0
MOVGT   R4, #1
```

```
LDR     R3, =Own_Tracked_Alt_Rate
LDR     R3, [R3]
CMP.W   R3, #0x258
BGT     loc_800380
```

```
MOVS    R4, #0
B       loc_800368
```

```
LDR     R3, =Two_of_Three_Reports_Valid
LDR     R3, [R3]
CBZ     R3, loc_8000378
```

Figure 8: Live ranges of malicious chunks for input data injection on a collision avoidance program (dashed brackets).

might not be fully realized. The live ranges of the three input variables `Cur_Vertical_Sep`, `Own_Tracked_Alt_Rate`, and `Two_of_Three_Reports_Valid` lie between beginning of the program and the locations where they are first used, as represented by the dashed brackets. Details of the live ranges of other attacks in this paper are described in Section 5.

To find the live range of a malicious chunk, we perform a data flow analysis. More specifically, the data a malicious chunk intends to modify or retain is only available in some locations of the benign program. For example, if the malicious chunk wants to modify the data at a memory address associated with the input, the data is only available at the points between where the data is stored to the memory address and where the data is first used. Or if the malicious chunk wants to log the data at a memory address, this data is also only available in some locations. We perform data flow analysis on the benign program to locate where the data is created and where it is consumed or updated. In this way, we are able to identify a set of candidate locations where the data is available. We consider the set of these locations as the live range of the malicious chunk.

Once live ranges for all malicious chunks are identified, we employ an optimization algorithm based on Monte-Carlo Tree Search (MCTS) to find where in the live range to inject each chunk such that the program with all malicious chunks injected still has a high confidence score.

4.3 Generation of Evading Samples

We formulate finding an evading sample as an optimization problem where we want to maximize the confidence score of the program with the malicious code injected. As stated in Section 3, the detector has a confidence score threshold, determining whether the execution is benign (high confidence)

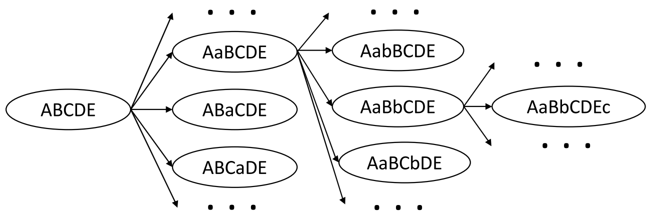


Figure 9: Search tree of the problem. The root node of the tree represents the benign program while the leaf nodes represents programs with all malware chunks injected.

or malicious (low confidence). Our optimization objective is to maximize the confidence score, which can account for the noise caused by different data values or measurement noise. This also improves the consistency of evasion for optimized samples. To imitate the strategy of an attacker generating samples iteratively, we model the process of finding the optimal injection combination as a tree search problem.

Specifically, we define the nodes in the tree to be an intermediate snapshot of the program with some malicious chunks injected, as seen in Figure 9. The root node represents the benign program while the leaf nodes represent the program with all malicious chunks injected. We call a program with all malicious chunks injected an injection combination. The edges in the tree represent the action of injecting the next malicious chunk. Starting from the root node, the malicious chunks are injected sequentially, one at a time, until the leaf nodes are reached. A malicious payload consists of 3 chunks a, b and c injected into a benign program $ABCDE$. The children of a node represent where the next malicious chunk can be injected. The choice of the child of a node is subjected to a set of constraints. Firstly, the malicious chunk can only be injected inside its live range. Secondly, malicious chunks must be executed in the same order as in the original malicious payload. This means if two chunks need to be executed one after the other, once the first chunk is injected to a location, any location before that are not available for the second chunk. Finally, all the malicious chunks need to be executed simultaneously in at least one execution path to ensure semantic preservation. Therefore, once a chunk is injected to a location, the rest of the chunks can only be injected to the sub-graph of the control flow graph of the benign program, starting from that location.

The goal of the tree search is to find an injection combination that gives the maximum confidence score

$$\max_{C} s \tag{2}$$

where C is an injection combination. The solution of the tree search problem can be accomplished with various methods such as the min-max algorithm, greedy search, or MCTS. The min-max algorithm is an exact method, enumerating all possible actions to find the optimal solution. However, as the program or malware size increases, the number of possible combinations explodes, making this method computationally infeasible. The greedy algorithm moves down the tree by picking up the child with the highest score, therefore, it is sub-optimal as in this way it won't be able to take longer term dependencies into consideration. MCTS can be considered a solution that balances between min-max and greedy search. For a node, it does not enumerate all possible actions, nor does it determine the value of this node without looking down the tree at all. MCTS estimates the value of a node by trying several possible solutions associated with this node. We propose using MCTS to solve the optimization problem.

Below we describe in detail how we adapt MCTS into our optimization scheme to find an evading malware. On a high level, our MCTS based optimization starts from a single root node, which is the benign program in our case. It explores and expands the tree iteratively by injecting malicious chunks one at a time. At the beginning, it explores randomly as there is not enough information about the tree. It evaluates the impact of injecting a malicious chunk at a candidate location on the confidence score, accumulating information over iterations. As it obtains more information about the how injecting the malicious chunks affects the confidence score, it starts to get an idea of which locations are better (resulting in higher confidence scores). It then exploits this learned knowledge to guide subsequent iterations exploring the tree more efficiently.

At each iteration, MCTS has four steps: selection, expansion, rollout and backpropagation.

Selection. In selection, MCTS starts from the root node and recursively traverses the child node with the highest value until a leaf node is reached. The leaf node represents the program with some malware chunks injected. We use the following formula [45] to compute the value of a node

$$\frac{s-\eta\sigma}{n}+c\sqrt{\frac{\ln N}{n}} \tag{3}$$

where s is the sum of the confidence scores of previous iterations. σ is the sum of the standard deviation (`stdev`) of the confidence scores of previous iterations as well. We will elaborate more on these two terms below. η is a coefficient balancing the two terms. n is the total number of visits of the selected node and N is the total number of visits of its parent node of the last iteration. Equation 3 helps find an injected program with high confidence score and low variation.

During our experiment, we observe that the confidence score of a power side-channel signal can vary a lot even for the same version of the program with malicious chunks injected. This is because injecting the malicious chunks changes the signal patterns. As a consequence, the signal moves to a much more nonlinear location in the feature space. At this location a slight change in the signal (possibly caused by measurement noise or different data values) can result in a significantly different confidence score. A more robust evading sample needs to be less sensitive to noise, or in other words, have low variance. Therefore, we penalize evading samples with high variation by including the σ term in Equation 3.

Equation 3 balances the search between exploration and exploitation. The first term controls exploitation, it indicates the average confidence score of all the injection combinations (associated with this node) tried so far. The second term controls exploration, it is large for nodes that have not been explored many times. Therefore, the algorithm tends to choose unexplored nodes. As the number of iterations increases, all nodes are explored at least once. Therefore, the first term starts to take control. This means the model starts to shift from exploration to exploitation.

Expansion. After a node is selected, the tree will be expanded from this node. A child of this node is created. In our attack, this essentially means injecting the next malware chunk. The choice of which child to create is random.

Rollout. Starting from the node created above, a rollout is carried out by repeatedly injecting the remaining malicious chunks randomly until all chunks are injected. After all chunks are injected, that version of the program is compiled and uploaded to the device. Side channel signals are then collected during the execution. Usually CPS applications execute only one or a few possible execution paths during normal operation, so we only collect side channel signals that correspond to these paths.

Side channel signals of various test inputs are collected to take data differences into consideration (Section 2). In addition, multiple signals are collected for each test input to account for circuit noise. Collected signals are sent to the side channel detector and a confidence score is obtained for each signal. Mean and `stdev` of the the confidence scores are then computed.

Backpropagation. Finally, the values of all nodes involved in this iteration of the optimization are updated based on the Rollout result, i.e., the mean and `stdev` computed in the Rollout step are summed into s and σ in Equation 3 respectively. n and N are also incremented accordingly.

The four steps above are repeated until a predefined iteration limit is reached. For each iteration, the injected program generated in the Rollout step is saved together with its corresponding confidence scores. The iterations will also stop if no improvement can be observed over multiple iterations. After the search process is done, injected program with the highest average confidence score is chosen as the result.

During the optimization process, our MCTS based algorithm accumulates information about injecting malicious chunks over multiple steps instead of only considering the instant impact of one single chunk. Therefore, it can estimate the impact of a injection on the confidence score better. As the algorithm learns better estimates about the values of injecting different malicious chunks at different candidate locations, it will visit those with a higher value more frequently in subsequent iterations. In this way, it can find an injection combination with high confidence score more efficiently.

Name	Description	# paths	≈ # instr.
path	Robotic automated arm path planning	8	200
collision	Siemens aircraft altitude control module	23	300
cruise	From Crazyflie drone altitude control	8	600
kalman	From Crazyfile drone state estimation	4	1250
particle	From Udacity's self driving car simulator	16	3000

Table 1: Control programs used in our evaluation.

5 Evaluation

5.1 Experimental Setup

Hardware Setup. We use a NEWAE CW308T-STM32F3 (with an ARM Cortex M4 microprocessor) as our target embedded platform, as the Cortex M4 is designed for cyber-physical applications as well as IoT systems [1]. Power side channel signals are collected using a NEWAE Chipwhisperer-Lite CW1173 oscilloscope[2]. The CW308T-STM32F3 board contains built-in circuits for measuring power side-channel signals from the power pin of the microprocessor. The trigger signal is sent out from one of the I/O pins. Both the power signal measurement and the trigger signal are passed to the Chipwhisperer. We modify the Chipwhisperer software to align and truncate multiple measurements based on the trigger signal. The microprocessor runs at a clock speed of 10MHz. The sampling rate of Chipwhisperer is 4 times the clock speed of the microprocessor.

Target Programs. We use a variety of programs as our target for evaluating the proposed attack. These programs are extracted from different control applications designed for aircraft, drones, and autonomous vehicles. Table 1 describes the number of control paths and instructions of these programs. For the cruise control, Kalman filter, and particle filter programs, the floating point unit on the target board is enabled. For simplicity, and because the ARM Cortex M4 does not have address-space layout randomization, we hard-code the addresses in our malware when possible.

Constructing a Side-channel Detector. To construct a power side-channel based anomaly detector for a given target program, we first identify all execution paths of the program [22]. Then for each path, we generate five test cases. We collect multiple power side-channel signals for each of these test cases, representing all possible behaviors of the program. The detector is then trained using these signals. We consider three detectors that are used in prior works: 1. a Bidirectional Long Short-Term Memory neural network [22] (BiLSTM), 2. a Bidirectional Recurrent Neural Network (BiRNN), and 3. a Hidden Markov Model [32] (HMM). We implement (1) and (2) based on available code, and implement (3) closely following its description in previous work. Our reproductions

[2](https://wiki.newae.com/Main_Page)

of prior works have comparable accuracy with respect to the original schemes as shown in our evaluation.

All three detectors operate on signal segments. Signal segments are extracted using a 90% overlapping sliding window on the power side-channel signals [22]. For the HMM, the signal segments were used as observations. The HMM state is defined as the unique samples in the observation set. Multiple HMM models are trained. Each one corresponds to one execution path and is trained using power side-channel signals of that path. To classify a test signal, all the trained HMMs are queried using the signal, and each HMM produces a log-likelihood score that represents how likely the signal is generated by the HMM. The HMM (or the corresponding execution path) with the highest log-likelihood score is the prediction. The overall confidence score of the multi-HMM model is the highest log-likelihood score.

Window Size and Features. For both time and frequency [22] representation of the window, we test different window sizes. Our goal is to identify values that allow us to reach comparable detection performance to prior work (see Table 2). In particular, we tested multiple window sizes starting from a base window of 4 signal points (representing a single instruction), progressively increasing the window size until matching the prior work performance. We note that the window sizes of all the detectors stay within the control loop (scan cycle). We further discuss the possibility of windows spanning multiple control loops in Section 6.

Detector Training. We train a detector with the power side-channel signal and its corresponding execution path pairs. We collect 20 signals for each test case as the training dataset, another 10 as the validation dataset. For the BiLSTM and BiRNN, the average number of epochs during training is 1000. For the HMM, we use a max of 200 iterations and a tolerance of 0.01. We empirically determine the architecture of the detector for each target program, i.e., size and number of hidden layers, by starting with a hidden size of 64 and 2 layers, and increasing the numbers until we achieve both a good validation classification accuracy as well as high confidence scores of all signals in the validation dataset. We use a similar strategy for the HMM with a starting point of 10 states and one mixture component.

Detector Accuracy. We report the classification accuracy and the area under the ROC curve (AUC) of each detector on its corresponding testing dataset in Table 2. Both BiLSTM and and BiRNN have good classification accuracy and detection performance for all the programs with both time and frequency features. The results match with the detection solutions in related work [22, 32]. Note that the results of HMM are not as good as others. We spent a lot of effort tuning the parameters of these HMM models but were not able

BiLSTM						
Feature		Time			Frequency	
Metric	WS	Acc.	AUC	WS	Acc.	AUC
path	4	98.01%	0.99	8	99.02%	0.99
collision	8	100.0%	0.99	16	100.0%	0.99
cruise	8	100.0%	0.99	16	100.0%	0.98
kalman	4	99.60%	0.99	8	100.0%	0.98
particle	64	99.86%	0.99	64	99.98%	0.99
BiRNN						
Feature		Time			Frequency	
Metric	WS	Acc.	AUC	WS	Acc.	AUC
path	8	99.26%	0.99	8	100.0%	0.98
collision	8	100.0%	0.98	16	100.0%	0.98
cruise	8	99.17%	0.99	16	99.12%	0.99
kalman	8	99.57%	0.99	8	98.23%	0.98
particle	64	99.22%	0.98	64	98.99%	0.99
HMM						
Feature		Time			Frequency	
Metric	WS	Acc.	AUC	WS	Acc.	AUC
path	8	98.23%	0.99	8	99.91%	0.99
collision	8	99.16%	1.00	8	100.0%	0.98
cruise	8	99.01%	0.95	8	100.0%	0.96
kalman	8	77.50%	0.85	8	85.50%	0.82
particle	16	72.00%	0.76	16	63.50%	0.73

Table 2: Performance of detector models constructed for all programs with both Time and Frequency features, comparing Window Size (WS), Accuracy, and Area Under Curve (AUC).

to achieve optimal results for all the programs. We report the best results we obtained in the table. We speculate that (unlike neural network-based models) the HMM cannot deal with long sequences well. As the program becomes larger, the HMM can no longer capture the temporal dependencies in the signal well. From the adversarial perspective, attacking a weaker detector is easier.

Exploratory Analysis: Instruction Insertion. Before we present our evaluation results on the real control attacks, we perform an exploratory analysis on the trained detectors to estimate the impact that deviations from a benign program introduce in the detector output. We consider both inserting and altering instructions.

For instruction insertion, we create a chunk consisting of several NOP instructions. We vary the number of NOP instructions in the chunk and insert it into the beginning, the middle and the end of the benign program. We find that, without the optimization proposed in this paper, even a single instruction insertion can lead to detection. For example, in the collision avoidance program, the insertion significantly decreases the output confidence score of the detector from an average of 0.99 to 0.87. This is because the side-channel detector captures the temporal relations of the signal segments (time or frequency features) of benign samples. Inserting new instructions introduces new signal features as well as shifts the rest of the signal points from the injection point. This has the effect of altering the internal states (i.e., hidden states) of all the affected signal segments. Such alteration accumulates along the sequence, making the final output of the detector deviate significantly from normal values (values of benign samples). However, we will show below that with our proposed attack, it is possible to find locations in the benign program to inject malicious chunks such that the internal states can be ’routed’ back to normal. In this way, the malicious sample stays undetected. We did not investigate the insertion of very long malicious instruction sequences, but for all the evaluated representative malware (some up to half the size of the benign program), our approach is able to find evading samples.

Exploratory Analysis: Instruction Alteration. The goal of instruction alteration is to replace individual benign instructions with others that have malicious effects, but do not impact the detector’s classification. In addition, after the alteration the semantic and data flow of the original program should stay unchanged—otherwise the program might crash or misbehave in a noticeable way. This means keeping the original program’s behavior as intact as possible while achieving the desired side-effects of the malware, which imposes a large number of hard constraints. Therefore, we only perform some preliminary studies. We consider altering an arithmetic or load instruction by instead moving the result to the register directly. We choose one instruction from the beginning, the middle and the end of the program. We find that the average confidence score suffers a detectable drop from an average of 0.99 to 0.93. This is because altering an instruction changes the signal profile of itself and the surrounding instructions, causing internal states of the detector to deviate from the normal state. A search for more suitable alteration would require more complex semantic constraints and a much larger search space compared to instruction insertion only. As a result, in the remainder of this work we focus on instruction insertion.

Training the Adversary’s Substitute Setup. Normally, training for a side-channel detector is done in-situ with training traces being acquired during normal operation. In the case of the substitute setup, the adversary does not have a complete operational system for training, instead knowing only the control program and hardware platform. To compensate, the adversary can generate a set of program inputs with thorough path coverage using a symbolic execution framework like Angr [59] or KLEE [8]. These program analysis frameworks are flexible tools for verifying program correctness, finding bugs, and exploring control flows.

Running Angr on a binary program produces a set of logical expressions containing the path constraints for each control path within a program. The path constraints determine input variable value ranges that will lead execution down a certain control flow. We then evaluate these logic expressions using a logic solver (such as Z3 [12]) to generate multiple unique concrete inputs for each control path. This allows us to train our substitute classifier against a wide array of scenarios, representative of normal operation. While in general, symbolic execution faces issues with exponential growth of paths, it remains feasible for size-limited embedded programs.

Avg # Inst. per chunk / Total # Inst.	path	collision	cruise	kalman	particle
False Data Injection	4/36	4/48	4/24	4/16	4/28
Overwrite Actuation Output	4/12	4/12	4/16	4/12	4/8
Control Parameter Attack	4/16	4/16	4/12	4/24	4/28
Data Logging	8/32	8/64	8/56	8/72	8/64
Confidential Information Disclosure	7/119	7/119	7/119	7/119	7/119

Table 3: The instruction count (by clock cycle) of malware chunks/samples.

5.2 Evaluation of Attack

We now evaluate our attack using five different malware payloads. The payloads range from few instructions customized for the target program (e.g., Overwrite Actuation Output) to generic longer sequences of 119 instructions in the case of Confidential Information Disclosure. When implementing the attacks, we first disassemble the program binary. Then, for each specific type of attack, we determine the data that we want to attack, and construct malware chunks based on this. We run data flow analysis afterwards to determine the life cycle of each malware chunk. Finally, after the malicious chunks are injected, we re-assemble the assembly code into a binary. We then collect power side-channel signals during the program execution. Those steps are repeated for every iteration of the proposed attack framework. To construct malware chunks, we only use registers that are not used by the target program to avoid potential conflicts.

The average number of instructions per chunk and the total number of instructions of a malware sample for all 5 types of malware are shown in Table 3. For example, 9 chunks of malware (with average size 4 instructions) were used for False Data Injection on the path program.

False Data Injection/Overwriting Actuation Outputs. These two classes of attacks modify the inputs and outputs of the program, resulting in malicious behavior. To implement this type of attack, we first reverse-engineer the binary of the target program and locate the memory addresses of where input/output data are stored, then refer to those addresses in our malicious code to modify the corresponding values.

We consider the modification of a single input/output variable to be one malware chunk. The live range of each chunk is constrained to be between the beginning of the scan cycle of the program and where the data is first used (input) or between the data is last written and the end of the scan cycle (output). By serving false data to the program, the attacker can control the physical system arbitrarily.

BiLSTM										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	1.0009	1.0036	1.0014	0.9952	1.0039	1.0031	0.9973	0.9975	1.0001	1.0045
collision	0.9985	0.9985	1.0023	1.0004	0.9995	1.003	1.0028	0.9986	0.999	0.9978
cruise	1.005	1.0019	0.9967	0.9988	0.9978	0.9987	0.997	0.9965	0.9973	0.9961
kalman	1.0027	0.9954	0.9951	0.9954	1.0042	1.004	0.9952	1.0045	1.0047	1.0047
particle	0.997	0.9978	1.0006	1.0045	1.0007	0.9986	0.998	1.0044	0.9989	1.0033
BiRNN										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9973	0.9977	0.9961	0.9987	0.9995	1.0026	1.0048	1.0023	0.9997	0.9964
collision	0.9975	0.9966	0.9982	0.9998	0.9996	0.9953	1.0008	1.0033	1.0045	0.9973
cruise	0.9978	0.9956	1.0038	0.9984	0.9996	0.9981	0.9954	1.0004	1.0026	1.0023
kalman	0.9951	0.9994	1.0004	1.0022	1.0045	0.9956	1.0037	1.0006	1.0033	0.9952
particle	1.0026	0.9954	0.9956	0.9974	1.0011	0.9994	0.999	0.999	1.0034	0.9979
HMM										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9993	0.9987	0.9969	0.9970	1.0035	1.0020	1.0007	0.9995	0.9975	1.0008
collision	1.0024	1.0032	1.0038	1.0013	0.9982	1.0027	0.9957	0.9984	0.9963	0.9998
cruise	1.0039	1.0038	1.0018	1.0020	0.9953	0.9951	1.0024	1.0003	1.0027	0.9969
kalman	1.1398	1.2010	1.0389	1.1015	1.1290	1.2312	1.1026	1.1107	1.1114	1.1023
particle	1.1530	1.1115	1.1128	1.1227	1.1511	1.1199	1.1392	1.1143	1.1879	1.1501

Table 4: Normalized average confidence scores (attack performance) on the tested programs with respect to different attacks.

Control Parameter Attack. The control parameter attack seeks to change memory locations containing constants or parameters used by the program. Similar to the previous attack, the control parameter attack is also implemented by reverse-engineering the binary to locate the memory addresses that correspond to the control parameters, manipulating those addresses in the malicious code. In this type of attack, the modification of each control parameter is treated as one malware chunk. The live range of each chunk is between the beginning of the program to where the corresponding control parameter is first used. Attacking the control parameters can effectively drive the physical system into dangerous states, potentially causing catastrophic damage such as crash.

Data Logging. This attack stores data to a predetermined location in memory, allowing it to be retrieved later or reused in a common replay attack scenario. Such an attack is implemented by first reverse-engineering the program binary to identify the addresses of the data that to be logged. The malicious code fetches the values of those addresses and store them in unused addresses in the SRAM. The modification of each data (address) is considered one malware chunk. The live range of these malware chunks is constrained by which locations in the code the desired data are available. Data logging can essentially steal sensitive data from the target program.

Confidential Information Disclosure. This attack scenario involves leaking confidential data used by the program over a communication channel that is available to the attacker. This type of attack is very similar to data logging with the difference being that the data is sent out in real time. In our implementation, we choose UART as the channel, leveraging the UART transmission function in the CMSIS library[3]. We linearize the code by removing functions, condition checks and other branch instructions, treating each basic block as one malware chunk. Again, the live ranges of the malware chunks are subject to the availability of the data to be sent out.

Research Questions. We aim to answer the following research questions by evaluating the proposed attack:

- RQ1: Can the proposed attack framework help find evasive malware sample injection strategies?
- RQ2: Can the proposed attack framework work under a cross-device setup for real-world settings?
- RQ3: Can the proposed attack framework even work under a cross-model setup for adversaries without access to the target detection models?
- RQ4: Can the proposed attack be still effective when adversarial examples are included during the training of the detectors?

RQ1: To answer RQ1, we apply our proposed attack framework to all the aforementioned attack types against the 5 control programs. We present the highest average confidence score achieved for each program and attack pair in Table 4. The reported confidence scores are normalized by the average confidence scores of benign traces. Therefore, a score close to or greater than one indicates a successful attack. As can

[3]The CMSIS library is a HAL implementation for ARM Cortex.

be seen from the table, the proposed attack is able to find an evading sample with very high score for all the scenarios. By splitting the malicious payload into small chunks and optimizing to find locations in the benign program to inject the chunks, the proposed attack framework manages to hide a malicious payload in a benign program. For HMM-based detectors that are weaker (e.g. kalman and particle), since the average confidence score of the benign signals are lower, the proposed attack can even find evading samples with a significantly larger confidence score. As mentioned above, injecting malicious code deviates internal states of the detector. The optimization of our proposed attack finds locations in the benign program to inject each chunk such that the internal states can be 'routed' back to normal. In this way, the malicious sample stays undetected.

We also compare the proposed attack framework with a baseline attack strategy, i.e., injecting the malicious payload directly without our optimization algorithm (represented by BASELINE). For this baseline attack, we manually identify a location in the benign program where injecting the malicious payload will be effective. For example, in the false data injection attack, we inject the malicious payload at the beginning of the program. We use CHUNK to represent the proposed attack approach in which payload chunks are injected into locations in the program determined by our search algorithm.

Figure 10 shows the mean of the normalized scores achieved for each strategy for a BiLSTM-based detector using time feature. For each attack, the scores are averaged over all the five programs. As shown, BASELINE results in lower scores compared to CHUNK, as injecting the malicious payload introduces signal patterns which cannot be recognized by the detector. Therefore, the output confidence score is still low. However, in the case of CHUNK, our search algorithm manages to find the optimal locations to inject the malicious chunks. In this way, the confidence score is kept high.

The proposed attack helps find a more robust evading sample. As is mentioned in Section 4, we observe that for a program with a malicious payload injected, the confidence scores tend to have a large amount of variation. Based on Equation 3, the optimization aims at minimizing this variation. This can help find a more robust evading sample. We apply the attack approach again on all the five programs with respect to all five control attacks. But this time, we do not consider the standard deviation in the formula, i.e., $\eta = 0$. We report the mean and max/min range of the confidence scores in Figure 11. The results are averaged over all the five programs. Again a BiLSTM-based detector using time feature is used.

As illustrated, without considering the variation, the attack approach cannot find an evading sample with low variation for most of the control attacks. Only by taking the variation into account in the optimization objective can the attack approach can find evading samples with low variation, i.e., a more robust evading sample. This is because without considering the variation of the confidence score in the optimization formula,

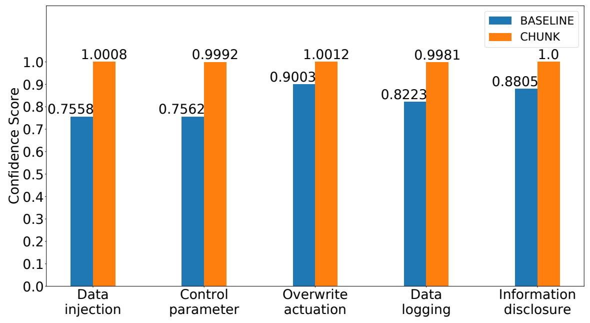


Figure 10: Comparison of attack strategies (averaged over all the five programs). BASELINE represents injecting the malicious payload as a whole. CHUNK represents injecting payload chunks in optimized locations.

the found evading samples might fall into a very sensitive (nonlinear) region in the signal space.

Therefore, a tiny change in the signal due to circuit noise or data differences might alter the confidence score significantly. Such a result is not robust. With the variation of the confidence score plugged in the formula, our search algorithm looks for samples with both a high score and low variation. Therefore, the result found is more robust and less likely to be detected.

5.3 Detailed cross-device attack performance

We report the detailed cross-device attack performance for all the programs and with respect to different attacks in Table 5. The collision avoidance and particle filter program have slightly lower confidence scores over all the attacks compared to other control programs. This is because these two programs have larger number of classes compared to other control programs. Therefore, their loss surfaces are more complicated. This makes them more sensitive to changes in the input.

RQ2: As described in our attack model, realistically the attacker cannot query the target microprocessor unlimited number of times. Instead, they can construct a substitute setup to generate malware samples and apply them to the original microprocessor. To answer RQ2, we assume the attacker has a substitute setup for generating the malware samples. Specifically, they have a substitute device which is exactly the same as the target one, and a copy of the program binary. They can upload the program binary to the substitute devices and collect power side-channel signals. We also assume the attacker knows the model architecture of the detector and they can train a substitute detector using the signals they collected. After generating the malware samples, the attacker can apply them to the original device under attack.

We compute the normalized confidence scores for all the programs with respect to different attacks. We obtain an average score of approximately 0.93. Detailed results can be found in Table 5. This is slightly lower than on the original

BiLSTM										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9361	0.9385	0.9128	0.9449	0.9239	0.9376	0.9333	0.924	0.9247	0.9215
collision	0.9199	0.9086	0.9201	0.9048	0.919	0.9093	0.9239	0.9049	0.9136	0.9231
cruise	0.9339	0.9339	0.9311	0.9159	0.9276	0.9263	0.9475	0.9383	0.9161	0.9105
kalman	0.9414	0.9367	0.9266	0.9496	0.9225	0.9164	0.9176	0.9272	0.9275	0.935
particle	0.9205	0.9172	0.902	0.9125	0.9042	0.9145	0.9222	0.9124	0.9149	0.9201
BiRNN										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9306	0.9249	0.9105	0.9354	0.9199	0.9403	0.9453	0.9485	0.9257	0.9321
collision	0.9176	0.9065	0.9181	0.9122	0.9253	0.9263	0.9236	0.9283	0.9105	0.9086
cruise	0.947	0.9188	0.9298	0.9361	0.9188	0.9138	0.91	0.9471	0.949	0.9269
kalman	0.9168	0.9316	0.9382	0.9364	0.9436	0.9421	0.9492	0.9331	0.9154	0.9147
particle	0.9103	0.9153	0.9272	0.913	0.9286	0.9164	0.9079	0.916	0.9259	0.9455
HMM										
Program	Input manipulation		Control parameter		Overwrite actuation		Data logging		Information disclosure	
Feature	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency	Time	Frequency
path	0.9208	0.921	0.9202	0.9185	0.9193	0.9234	0.921	0.9214	0.9244	0.9199
collision	0.916	0.9231	0.9202	0.9153	0.9157	0.9184	0.92	0.9152	0.9192	0.9202
cruise	0.9165	0.9206	0.916	0.9196	0.9217	0.9237	0.9178	0.924	0.9152	0.9159
kalman	1.0128	1.0113	1.0012	1.0102	1.1	1.1101	1.0007	1.0154	1.0321	1.0508
particle	1.0111	1.0203	1.0189	1.0169	1.0087	1.0147	1.0194	1.0155	1.0146	1.0128

Table 5: Normalized average confidence scores for cross-device attack performance on the tested programs with respect to different attacks. Here, the attack evaluation device is different from the attack optimization (substitute setup) device.

device. However, they are still acceptably high. This means the evading samples found by the proposed attack framework manage to transfer to a different device. We attribute the cross-device transferability of the proposed attack to the goal of the optimization. Even though on a different device the signal patterns can be different due to circuit noise, since the goal of the proposed attack is to find an evading sample with low standard deviation, the effect of such signal pattern difference does not significantly decrease the confidence score. We also note that the collision avoidance and particle filter program have slightly lower confidence scores over all the attacks compared to other control programs. This is because these two programs have larger number of classes compared to other control programs. Therefore, their loss surfaces are more complicated. This makes them more sensitive to changes in the input.

RQ3: RQ3 addresses questions of transferability by further restricting the attacker's knowledge and capabilities on the target; not only does the attacker have no access to the target microprocessor, nor do they have knowledge of the model of the detector. This can be the model type, the model architecture (e.g. number of hidden layers or the size of each hidden layer). We show that by exploiting the transferability of machine learning models, such an attack is still possible. We examine the transferability among the aforementioned three detector models as well as a substitute detector for BiLSTM

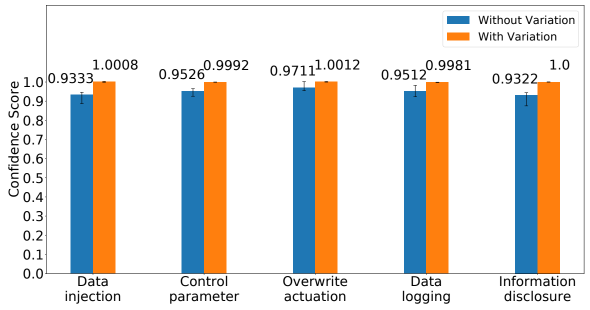


Figure 11: Comparison of the attack with and without considering the confidence score variance. The error bars represent the max and min confidence scores . By considering the variance, we find robust evading samples.

($BiLSTM_{sub}$). For $BiLSTM_{sub}$ we make the hidden layer size half and double the number of layers. These four models represent three different levels of transferability: cross different architectures of the same model, cross different models of the same family (i.e., neural networks) and cross different families of models. The detectors are trained using the power signals collected from the substitute device. We upload the evading samples found in RQ1 to the original device. We then collect 100 power side-channel signals for each execution path of the program from the original device. These signals are sent to the detectors to generate the confidence scores.

We report the normalized average score for each pair of

	Detector	BiLSTM	BiLSTM$_{sub}$	BiRNN	HMM
Time	BiLSTM	-	0.9323	0.9301	0.7712
	BiLSTM$_{sub}$	0.9301	-	0.9277	0.7513
	BiRNN	0.9135	0.9126	-	0.7912
	HMM	0.8012	0.8071	0.7908	-
Frequency	BiLSTM	-	0.9302	0.9287	0.8124
	BiLSTM$_{sub}$	0.9226	-	0.9191	0.7811
	BiRNN	0.9107	0.9097	-	0.5511
	HMM	0.7798	0.8025	0.8175	-

Table 6: Transferability between different models. Normalized average scores between each pair of models.

four models in Table 6. The reported results are averaged over all the programs and attack types for each model. As can be seen, cross architecture and cross model attacks transfer pretty well. This is due to the fact that these neural network-based models are trained with the same dataset, perform similar computations, follow similar optimization procedure, so loss surface of these models are similar to some extent. On the other hand, cross family attack does not transfer very well. This is because two families of models, such as neural networks and HMM, can be very distinct. To sum up, recall our attack model, an adversary only needs to know the family of the detector model to successfully perform the attack.

RQ4: RQ4 even further challenges the capability of the proposed attack approach by asking: *Is the proposed attack still effective even when generated evading samples are included in the training dataset of the detector?* To answer the question, we adopt adversarial training in the design of the detectors and test the proposed attack approach on them. Specifically, for a detector under attack, we create one more adversarial class apart from its original classes. To prepare data for training this augmented detector, we use the proposed attack approach to generate evading samples for all the attack types we considered. We pick the top 10 evading samples (in the sense of average confidence score) and collect power side-channel signals of these evading samples as signal samples in the adversarial class. This is to show the detector what evading malicious signals look like. We then train the detector in a standard way. Finally, we apply the proposed attack on the augmented detectors. To make the proposed attack compatible with the augmented attacks, borrowing from the idea from adversarial machine learning [28], we replace the confidence scores in Equation 3 with the cross entropy loss of the input signals with respect to the adversarial class. By maximizing this loss, we are essentially reducing the probability of classifying a malicious signal into the adversarial class. Once we obtain the result from the attack procedure, we classify the corresponding power side-channel signals using the augmented detector. We measure the successful rate of the attack by looking at how many signals are classified into any class other than the adversarial class.

	path	collision	cruise	kalman	particle
Time	99.78%	100%	97.13%	99.24%	98.55%
Frequency	99.26%	99.16%	98.01%	98.26%	97.98%

Table 7: Attack success rate for adversarial training augmented BiLSTM detector.

We perform the aforementioned experiment on BiLSTM-based detectors for all the five programs. To construct an augmented detector and measure the attack performance, evading samples of all the attacks we considered are used. We collect five power side-channel signals for each test input. We observe that all augmented detectors has very good performance (average classification accuracy 98%). For the attack, we report the percentage of signals not classified into the adversarial class as attack successful rate. The results are shown in Table 7. As can be seen from the table, even when evading samples are added to the classifier, our proposed attack can still find evading samples that can cause most of the power signals to be misclassified as benign. This is because data driven models such as a side-channel detector can easily suffer from lack of sufficient data, leaving a lot of corner cases near the decision boundary. The iterative MCTs optimization of our proposed attack approach can maneuver around the decision boundary, moves towards these corner cases (i.e., regions with a high loss with respect to the adversarial class). Therefore, our proposed attack approach can find an evading sample with a high attack successful rate. There is no direct way of compare the performance with and without adversarial training. This is because the definition of attack success rate is the percentage of signal samples not classified as the adversarial class. In the absence of adversarial training there is no adversarial class, and it is not possible to compute the attack success rate. However, as shown in Table 4 and Table 7, our proposed attack is successful in both scenarios.

6 Discussion

Improving malicious hit-rate. One limitation of this work as it stands is that a solution is not guaranteed even though we did not face this problem in our extensive experiments. Due to the constrained nature of the problem, we are not theoretically guaranteed to find a suitable evading example. To address this issue, one possible future direction is the incorporation of code randomization [25] applied to malware samples.

Recommendations for defense. A common strategy for mitigating the effect of adversarial examples is the use of adversarial training. Borrowing ideas from adversarial machine learning and including adversarial examples generated by our method in the training data can potentially produce a more robust side-channel monitor. However, our experiments indicate such mitigation is not effective against our proposed attack. Alternatively, the detector could perform information fusion and utilize power and electromagnetic side-channels at the same time. Misleading such a hybrid detection engine could be more challenging in practice.

As noted in Section 5.1, window sizes of all the detectors stay within the control loop (scan cycle). As a result, windows will never span two control loops at once. A possible avenue for detector improvements would be to use windows spanning multiple control loops. However, including more scan cycles in the window would increase the number of path combinations exponentially and potentially cause a state explosion. Given that the focus of this work was not on developing better detectors, we leave this exploration for future work.

Possible Transferability to EM Side Channel. The proposed attack approach can potentially be transferred to EM side channel for the following reasons. First of all, both power and EM side-channels originate from the same circuit components. Therefore, they may have high correlation. Moreover, the proposed attack approach is a search based attack; it does not depend on a specific type of physical side-channel signals to work. Therefore, it can also be applied to EM side-channel signal based detectors.

7 Related Work

Physical Side Channel Analysis. Many recent works analyze physical side channels (e.g. power consumption or EM radiation) produced by embedded microcontrollers to infer their internal behavior. Eisenbarth et al. [13] demonstrate the ability to reconstruct a program by modeling executed instructions with Hidden Markov Model (HMM) states. Other works leverage side-channel information to verify the integrity of code execution on a monitored device.

Liu et al. [32] track code execution using power signals. They measure the voltage drop over a resistor on the power pin of the monitored 8-bit AVR microcontroller. They train an HMM on normal program executions and use a maximum likelihood algorithm to detect abnormal control flows. Nazari et al. [39] measure EM spectra and use statistical tests to determine whether an execution follows a "benign" frequency distribution. Han et al. [22] use neural networks to classify benign and malicious EM signals. All of these works utilize data-driven models, which are susceptible to adversarial examples [19]. Our attack exploits the vulnerabilities of imperfect data-driven models to evade the detection mechanism.

Functional Malware Generation. The generation of functional malware to evade defensive classifiers has been previously studied in the case of PDF malware [11, 63]. They utilize mutation-based approaches such as genetic algorithms to convert benign samples to malicious-but-evasive samples while maintaining malicious functionality. In [11], the authors use a generation-testing loop to verify the functionality of a malware sample once it is generated. In [63], the authors use a query-based approach and knowledge of classifier features to guide their search for evasive and functional malware variants. We approach our problem similarly with exploratory methods to find evasive malicious samples with a query-based approach. In our problem, however, the features of the side-channel monitor are not readily mapped to the input domain. We further prune our search space using Monte-Carlo tree search and deep reinforcement learning.

Evading Classifiers with Adversarial Examples. Our attack is based on the vast amount of adversarial example research in machine learning [64]. A well known method in this domain is the fast gradient sign method (FGSM) for generating adversarial examples [19]. This method calculates gradients to determine the best perturbation direction to generate an adversarial example. The usage of gradient information is popular among many adversarial example construction techniques [37,47,64]. Even methods that do not use gradient information directly estimate gradient information using iterative and query-based techniques [26, 40]. In our case, the classifier uses physical signals created from discrete instructions executed on a processor. Therefore, in our case the classifier gradient does not contain direct information on how to create an adversarial example program. As a result, we use iterative gradient estimation techniques in the form of Monte-Carlo Tree search, as well as deep reinforcement learning to develop feasible adversarial examples.

8 Conclusion

In this work, we investigate the resilience of side-channel based control flow monitoring schemes. We design and demonstrate attacks that allow (despite the comprehensive monitoring enabled via power consumption) to *hide in plain sight*. Our attacks inject a functional malware payload; this is achieved using adversarial code injection and optimization techniques like Monte Carlo Tree Search to help explore the discrete and constrained state space. Our experiments on the popular ARM microcontroller demonstrate that even though side-channel monitors are physically isolated, measure immutable signals, use uninterpretable models, and present a highly constrained input domain for an attacker, they remain vulnerable to malicious code execution. Our results demonstrate that, although side-channel monitoring systems are well matched for real-time embedded control systems, they are not as secure as previous work would suggest. Therefore, more research is required to improve their robustness against evasive advanced adversaries, including (but not restricted to) the use of information fusion.

Acknowledgement

We would like to thank the National Science Foundation (NSF) for their support of our work through grants by the Cyber-Physical Systems (CPS) and Secure and Trustworthy Cyberspace (SaTC) programs.

References

[1] Arm cortex m4. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4. Accessed: 2021-06-01.

[2] Network defense. https://crypto.stanford.edu/cs155old/cs155-spring06/13-network-defense.pdf. Accessed: 2019-03-20.

[3] Pfp cybersecurity. https://www.pfpcyber.com/. Accessed: 2019-06-26.

[4] S. Abe, M. Fujimoto, S. Horata, Y. Uchida, and T. Mitsunaga. Security threats of internet-reachable ics. In *2016 55th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)*, pages 750–755. IEEE, 2016.

[5] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik. C-flat: control-flow attestation for embedded systems software. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 743–754. ACM, 2016.

[6] M. Balduzzi, C. T. Gimenez, D. Balzarotti, and E. Kirda. Automated discovery of parameter pollution vulnerabilities in web applications. In *NDSS*, 2011.

[7] L. Batina, S. Bhasin, D. Jap, and S. Picek. Csi nn: Reverse engineering of neural network architectures through electromagnetic side channel. In *28th USENIX Security Symposium (USENIX Security)*, pages 515–532, 2019.

[8] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[9] R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso. Zero-overhead profiling via em emanations. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 401–412. ACM, 2016.

[10] E. Chien, L. OMurchu, and N. Falliere. W32.Duqu - The precursor to the next Stuxnet. Technical report, Symantic Security Response, nov 2011.

[11] H. Dang, Y. Huang, and E.-C. Chang. Evading classifiers by morphing in the dark. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 119–133. ACM, 2017.

[12] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[13] T. Eisenbarth, C. Paar, and B. Weghenkel. *Building a Side Channel Based Disassembler*, pages 78–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[14] N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier. Technical report, Symantic Security Response, Oct. 2010.

[15] Y. Feng, O. Bastani, R. Martins, I. Dillig, and S. Anand. Automated synthesis of semantic malware signatures using maximum satisfiability. *arXiv preprint arXiv:1608.06254*, 2016.

[16] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. N. Choudhary. Towards online spam filtering in social networks. In *NDSS*, volume 12, pages 1–16, 2012.

[17] L. Garcia, F. Brasser, M. H. Cintuglu, A.-R. Sadeghi, O. A. Mohammed, and S. A. Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *NDSS*, 2017.

[18] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev. Ai 2: Safety and robustness certification of neural networks with abstract interpretation. In *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[19] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[20] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing. Lemna: Explaining deep learning based security applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 364–379. ACM, 2018.

[21] Y. Han, I. Christoudis, K. I. Diamantaras, S. Zonouz, and A. Petropulu. Side-channel-based code-execution monitoring systems: a survey. *IEEE Signal Processing Magazine*, 36(2):22–35, 2019.

[22] Y. Han, S. Etigowni, H. Liu, S. Zonouz, and A. Petropulu. Watch me, but don't touch me! contactless control flow monitoring via electromagnetic emanations. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1095–1108. ACM, 2017.

[23] K. E. Hemsley, E. Fisher, et al. History of industrial control system cyber incidents. Technical report, Idaho National Lab.(INL), Idaho Falls, ID (United States), 2018.

[24] S. Karnouskos. Stuxnet worm impact on industrial cyber-physical system security. In *IECON 2011-37th Annual Conference of the IEEE Industrial Electronics Society*, pages 4490–4494. IEEE, 2011.

[25] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis. Compiler-assisted code randomization. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 461–477. IEEE Computer Society, 2018.

[26] A. Kurakin, I. Goodfellow, and S. Bengio. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533*, 2016.

[27] J. Leyden. Polish teen derails tram after hacking train network. *Online at http://www.theregister.co.uk/2008/01/11/tram_hack/*, 2008.

[28] J. Li, F. Schmidt, and Z. Kolter. Adversarial camera stickers: A physical camera-based attack on deep learning systems. In *International Conference on Machine Learning*, pages 3896–3904. PMLR, 2019.

[29] Y. Li, M. Wang, C. Zhang, X. Chen, S. Yang, and Y. Liu. Finding cracks in shields: On the security of control flow integrity mechanisms. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1821–1835, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*, 2018.

[31] R. Lipovsky and A. Cherepanov. Blackenergy trojan strikes again: Attacks ukrainian electric power industry. *Online at http//www.welivesecurity.com/2016/01/04/blackenergy-trojan-strikes-again-attacksukrainian-electric-power-industry*, 2016.

[32] Y. Liu, L. Wei, Z. Zhou, K. Zhang, W. Xu, and Q. Xu. On code execution tracking via power side-channel. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1019–1031. ACM, 2016.

[33] L. Lu, R. Perdisci, and W. Lee. Surf: detecting and measuring search poisoning. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 467–476. ACM, 2011.

[34] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. *arXiv preprint arXiv:1612.04433*, 2016.

[35] S. E. McLaughlin, S. A. Zonouz, D. J. Pohly, and P. D. McDaniel. A trusted safety verifier for process controller code. In *NDSS*, volume 14, 2014.

[36] Y. Mirsky, T. Doitshman, Y. Elovici, and A. Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[37] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

[38] J. Mulder, M. Schwartz, M. Berg, J. R. Van Houten, J. Mario, M. A. K. Urrea, A. A. Clements, and J. Jacob. Weaselboard: zero-day exploit detection for programmable logic controllers. *Sandia report SAND2013-8274, Sandia national laboratories*, 2013.

[39] A. Nazari, N. Sehatbakhsh, M. Alam, A. Zajic, and M. Prvulovic. Eddie: Em-based detection of deviations in program execution. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 333–346. IEEE, 2017.

[40] A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[41] N. Papernot, P. McDaniel, and I. Goodfellow. Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*, 2016.

[42] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman. Sok: Security and privacy in machine learning. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 399–414. IEEE, 2018.

[43] D. Papp, Z. Ma, and L. Buttyan. Embedded systems security: Threats, vulnerabilities, and attack taxonomy. In *2015 13th Annual Conference on Privacy, Security and Trust (PST)*, pages 145–152. IEEE, 2015.

[44] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro. Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349. IEEE, 2020.

[45] E. Quiring, A. Maier, and K. Rieck. Misleading authorship attribution of source code using adversarial learning. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 479–496, 2019.

[46] M. S. Rahman, T.-K. Huang, H. V. Madhyastha, E. Wustrow, M. Faloutsos, J. A. Halderman, S. Abu-Nimeh, C. Kruegel, W. Lee, G. Vigna, et al. Efficient and scalable socware detection in online social networks. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, pages 663–678, 2012.

[47] J. Rauber, W. Brendel, and M. Bethge. Foolbox: A python toolbox to benchmark the robustness of machine learning models. *arXiv preprint arXiv:1707.04131*, 2017.

[48] M. T. Ribeiro, S. Singh, and C. Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144. ACM, 2016.

[49] J. Rrushi, H. Farhangi, C. Howey, K. Carmichael, and J. Dabell. A quantitative evaluation of the target selection of havex ics malware plugin.

[50] R. Rutledge, S. Park, H. Khan, A. Orso, M. Prvulovic, and A. Zajic. Zero-overhead path prediction with progressive symbolic execution. In *Proceedings of the 41st International Conference on Software Engineering*, pages 234–245. IEEE Press, 2019.

[51] H. Sak, A. W. Senior, and F. Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. 2014.

[52] R. Samani and C. Beek. Updated blackenergy trojan grows more powerful, 2016.

[53] Y. Shen, E. Mariconti, P. A. Vervier, and G. Stringhini. Tiresias: Predicting security events through deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 592–605. ACM, 2018.

[54] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. 2015.

[55] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. Sok: (state of) the art of war: Offensive techniques in binary analysis. 2016.

[56] C. Smutz and A. Stavrou. When a tree falls: Using diversity in ensemble classifiers to identify evasion in malware detectors. In *NDSS*, 2016.

[57] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting fuzzing through selective symbolic execution. 2016.

[58] O. Suciu, R. Mărginean, Y. Kaya, H. Daumé III, and T. Dumitraş. When does machine learning fail? generalized transferability for evasion and poisoning attacks. *arXiv preprint arXiv:1803.06975*, 2018.

[59] F. Wang and Y. Shoshitaishvili. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9. IEEE, 2017.

[60] S. Wang, K. Pei, J. Whitehouse, J. Yang, and S. Jana. Formal security analysis of neural networks using symbolic intervals. *arXiv preprint arXiv:1804.10829*, 2018.

[61] C. Whittaker, B. Ryner, and M. Nazif. Large-scale automatic classification of phishing pages. In *NDSS*, volume 10, page 2010, 2010.

[62] Y.-j. Xiao, W.-y. Xu, Z.-h. Jia, Z.-r. Ma, and D.-l. Qi. Nipad: a non-invasive power-based anomaly detection scheme for programmable logic controllers. *Frontiers of Information Technology & Electronic Engineering*, 18(4):519–534, 2017.

[63] W. Xu, Y. Qi, and D. Evans. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

[64] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE transactions on neural networks and learning systems*, 30(9):2805–2824, 2019.

[65] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1105–1116. ACM, 2014.