CS249B Final Project

Execution Flow Invariants Mining & Verification for CPIS

Bo Pang (pangb@berkeley.edu)
Spring 2021

Abstract - In this project, I introduce a workflow that allows the CPIS to verify the original CPS's software execution flow. My workflow abstracts the raw execution traces using counting vectors, finds linear invariants during the training phase, and verifies new traces against these invariants during regular operation of the CPIS. During my test using CPIS simulator, the addition of this execution flow verification brings the detection rate of man-in-the-middle attack from 54% to 89%.

OVERVIEW

Our capstone project builds a Cyber-Physical Immune System (CPIS) that works along with an ordinary cyber-physical system (CPS), collecting information non-intrusively and detecting anomalies in that CPS. So far, CPIS focus on the data domain (e.g., gather sensor data or data being exchanged in the CPS network). This data can be used to infer and verify the runtime states of the CPS in most situations. However, it is inadequate to identify some issues. E.g., the attacker can inject malicious executable into the CPS software and keep the runtime data unchanged while modifying the control behaviors.

In this project, I enhance our CPIS by also looking into the original CPS's software execution flow. The analysis of execution flow unveils the CPS software's actual behavior, making our CPIS more competent in detecting runtime anomalies caused by, e.g., network attacks and software injection.

Our existing CPIS design can obtain execution traces and data from each processor in the original CPS, leveraging the trace & data exporting hardware available on modern embedded processors. My work bases on that existing infrastructure. It enables the CPIS to (1) learn from the execution traces (aka. the training phase), assuming we do not have access to the source code in original CPS; and (2) verify the traces for anomaly detection (aka. the testing phase).

In the rest of this report, I will review some relevant works, give a detailed description of my proposed workflow, and then talk briefly about the test results.

RELEVANT WORKS

Based on embedded execution traces, I have seen a variety of workflow to abstract trace and detect execution flow anomalies.

Using system calls to abstract traces is frequently mentioned, according to my literature review. The basic idea is to filter the trace and only look for those involve kernel module invocations (aka. system calls) and use them as the representative of software states. Murtaza et al. verified the effectiveness of this abstraction using real-life examples [1]. Sekar et al. presented a workflow to build deterministic FSA using system call traces and validate incoming traces [2]. On top of that, Feng et al. dive deep into the runtime stack. By continuously monitoring the PC pointer and the return address being pushed into the stack upon each system call, their workflow can detect more indiscernible software anomalies, including modified return address caused by buffer overflow [3].

The idea of system-call-based trace analysis looks promising but may not apply to many embedded applications where the software stack doesn't involve a kernel or OS. Nonetheless, it is still possible to identify some special instructions and use them to filter the trace.

Zadeh et al. borrowed ideas from signal processing [4]. They treated live traces as time-series data streams and then applied Fourier transformation. Afterward, they looked into the frequency domain (spectrum) and developed multiple metrics for anomaly detection, using, i.e., peak frequency and DC significance. This method requires labeled data points and expects the trace to show periodic solid features. However, in CPIS, obtaining labeled data is not feasible due to the wide variety of anomalies or cyber-attacks.

Lu and Lysecky proposed an intrusion detection method that exams time elapses between instructions (or 'events' in their context) [5]. Since it is not feasible to keep track of each event, they also proposed a workflow to select and only track specific event pairs - whose timing characteristics are most helpful in revealing intrusions. Lu and Lysecky also proposed a hardware architecture that precisely measures the time elapses and can store & process the timing information. This work can be a good reference if I can utilize programmable hardware.

Nandi et al. proposed an algorithm to reconstruct a CFG from execution logs [6]. The algorithm involves finding nearest-neighbors in log sequences, identifying places where the software flow changes (fork or merge) and finding immediate predecessors & successors during CFG construction. Though log sequence can be very different than traces, especially in terms of quantity and consistency, their proposed algorithm can be a good reference if we go down the path of CFG reconstruction.

In terms of log analysis, Xu et al. mentioned mapping groups of log messages into "counting vectors", then process the vectors with PCA [7]. Lou et al. carried the similar idea further by mining linear invariants using counting vectors via SVD or Eigen decomposition [8]. Though their works are all based on log analysis, I believe the same concept and strategy can be applied to trace processing. These works inspired me to look into trace abstraction using counting vectors and then find linear invariants in those vectors.

My Workflow

I decide to utilize counting vectors to abstract traces, similar to [7][8]. For one, counting vectors compress the information in traces effectively, thus suitable for CPIS where the computing power is limited. For another, counting vectors from multiple synchronous executions can be combined and processed together. Based on counting vectors, I propose a workflow that finds linear invariants during the training phase and verifies new traces against these invariants. The workflow consists of three parts: counting vector filter, Invariants mining, and Invariants verification.

Part I: Counting Vector Filter
When the CPIS first attaches to a CPS node, it starts to see execution traces from that CPS node. Each trace entry usually contains the current program counter (PC) value or code line number if the compiler information is available.

Since we do not have access to the source code, CPIS will start from scratch and build a long "initial counting vector" covering every PC offset or line of code. An incoming trace entry triggers an increment of its corresponding counter. At this stage, we need to let the CPS node run extensively to increase code coverage as much as possible. If the CPS accepts configuration inputs, we will also need to run it under various inputs. *Figure 1 (LHS)* illustrates this step.

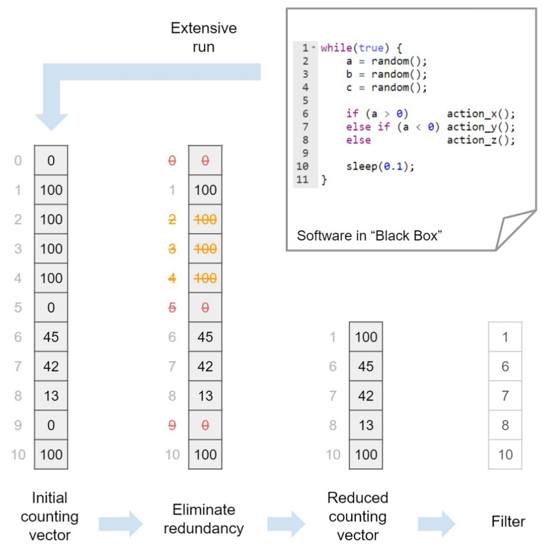

```
1  while(true) {
2      a = random();
3      b = random();
4      c = random();
5
6      if (a > 0)        action_x();
7      else if (a < 0)   action_y();
8      else              action_z();
9
10     sleep(0.1);
11 }
```

Figure 1: workflow to build a counting vector filter

After obtaining the "initial counting vector," the CPIS will scan the counting vector and eliminates redundant counters (or elements in the vector). Firstly, counters that have value equals the preceding elements will be removed, as they are likely associated with a consecutive execution path – the ROI of keeping track of those counters is small. Secondly, counters equal to zero will be removed too, as they probably associate with unreachable / inexecutable code regions or could not be covered by an extensive execution – either way, keeping track of them provides no benefits.

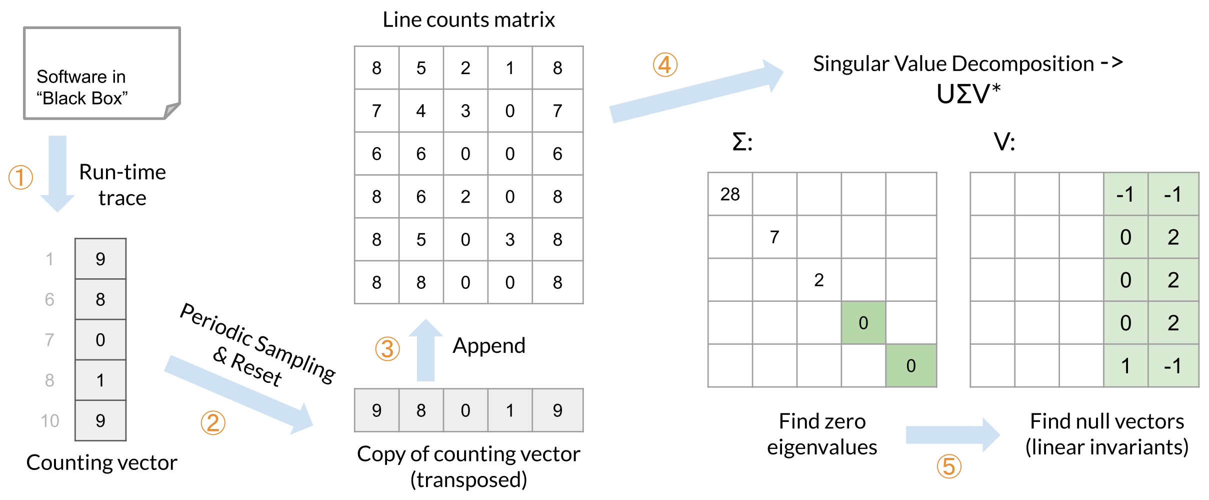
Finally, we obtained a "reduced counting vector." It serves as a filter - the PC offsets (or line numbers) associated with that vector are what the CPIS will look for in the next part. *Figure 1 (RHS)* illustrates the elimination of redundant counters and the derived filter.

Part II: Invariants Mining

With the filter obtained in the previous step, the CPIS now only needs to maintain a relatively small counting vector at a time. It can then find linear invariants from a bunch of such counting vectors. This process consists of 5 major steps, as shown in *Figure 2.*

Step 1: The CPIS will monitor the CPS during run time, gather traces, and increment a counter vector. Step 2 / 3: Periodically, the CPIS copies the counting vector, transposes it, and appends it to a counting matrix. The matrix with size M * N means it contains M counting vectors obtained previously, and each vector contains N counters (i.e., N different software instructions or lines of code).

After accumulating enough counting vectors for code coverage (e.g., there is no all-zero column in the matrix), the CPIS will execute a singular value decomposition (SVD) against that matrix, as step 4. The SVD finds the null space of the matrix, which corresponds to the linear relationships that apply to every counting vector. Finally (step 5), the CPIS



Software in “Black Box”
① Run-time trace
Counting vector
② Periodic Sampling & Reset
Line counts matrix
③ Append
Copy of counting vector (transposed)
④ Singular Value Decomposition -> UΣV*
Σ:
V:
Find zero eigenvalues
⑤
Find null vectors (linear invariants)

Figure 2: workflow to find invariants from a set of counting vectors.

extracts the singular vectors corresponds to zero singular values. These vectors are the execution flow invariants that our CPIS would use to verify incoming traces.

Part III: Invariants verification

After the CPIS derives a set of linear invariants, it can verify a newly received counting vector via matrix-vector multiplication and assert the result is an all-zero vector. To tolerant some miner error (i.e., the counting vector is sampled before the software compiling a loop), the CPIS can also look at the amount of failed assertion within a time window.

INTEGRATION AND TEST RESULTS

To test the proposed workflow, I implement the workflow in our CPIS simulator (the vehicle cruise control system). The workflow is hosted by the CPIS main node, where it will work together with the existing sensor data anomaly detector made in our capstone project. The counting vectors belonging to the two CPS nodes (cruise controller and engine controller) are built & reduced by two monitor nodes and send to the main node. The monitor extracts 9 counters from the engine controller and 4 from the cruise controller. The main node will concatenate both counting vectors into a single one (length of 13), given that the two CPS nodes execute in synchronized steps.

During the training phase (run the vehicle under varieties of throttle and speed combinations), the CPIS main node found 8 linear invariant vectors from the 13 counters. That enforces a good amount of execution path relationship between two CPS nodes, which is desired.

The man-in-the-middle attack is then used to test the effectiveness of verifying these execution flow invariants. The setup is the same as what we did in the capstone project to test the sensor data anomaly detector.

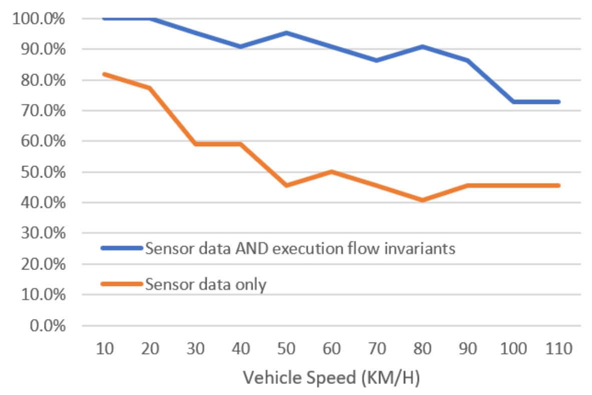


Figure 3: Detection Rate of MITM attack

The verification workflow raises the alarm when there are three consecutive violations of these invariants – it ensures zero false-positive under normal operations of the CPS.

The result is promising. With the addition of execution flow invariants verification, the CPIS can detect the man-in-the-middle attack within 10 seconds for 89% of the time instead of 54% before. *Figure 3* shows the detection rates under different vehicle speeds.

My experiment on CPIS achieves such improvement on attack detection rate is because the CPIS main node found a pair of execution flow branches that slow the vehicle down in both CPS nodes. The man-in-the-middle attack frequently breaks the link (i.e., the cruise controller wants to slow down but the engine controller does not) and thus being detected by the CPIS main node.

FUTURE WORK

An important question I did not address is how this method – especially finding invariants from counting vectors – can be affected by noise or imperfect counting vectors. When we sample a counting vector (step 2 in *Figure 2*) it is possible that the executable is not yet completing a cycle, yielding a counting vector that is not 100% representative of the accurate execution flow. More work needs to be done to investigate this problem and enhance my workflow.

References

[1] Murtaza, Syed Shariyar, et al. "A trace abstraction approach for host-based anomaly detection." *2015 IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA)*. IEEE, 2015.

[2] Sekar, R., et al. "A fast automaton-based method for detecting anomalous program behaviors." *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE, 2000.

[3] Feng, Henry Hanping, et al. "Anomaly detection using call stack information." *2003 Symposium on Security and Privacy, 2003.*. IEEE, 2003.

[4] Zadeh, Mohammad Mehdi Zeinali, et al. "SiPTA: Signal processing for trace-based anomaly detection." *2014 International Conference on Embedded Software (EMSOFT)*. IEEE, 2014.

[5] Lu, Sixing, and Roman Lysecky. "Time and sequence integrated runtime anomaly detection for embedded systems." *ACM Transactions on Embedded Computing Systems (TECS)* 17.2 (2017): 1-27.

[6] Nandi, Animesh, et al. "Anomaly detection using program control flow graph mining from execution logs." *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2016.

[7] Xu, Wei, et al. "Detecting large-scale system problems by mining console logs." *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009.

[8] Lou, Jian-Guang, et al. "Mining Invariants from Console Logs for System Problem Detection." *USENIX Annual Technical Conference*. 2010.