Code-Tampering Defense for Internet of Things Using System Call Traces

Rajesh Kumar Shrivastava(✉) and Chittaranjan Hota

Department of Computer Science, Birla Institute of Technology and Science, Pilani, Hyderabad Campus, Hyderabad, India
{p20150005,hota}@hyderabad.bits-pilani.ac.in

Abstract. This paper proposes a novel method to prevent an attack mounted by an adversary on an IoT device by executing suspicious system calls. An adversary in such cases would want to modify the behavior of an IoT device for hijacking the control by mounting malicious code. This paper uses system call traces to find out illegal accesses made on an IoT node. We develop a kernel-level processor tracing method for jeopardizing adversary's activities. The method is rigorously tested on various IoT nodes like Raspberry Pi 3, Intel Galileo Gen 2, Arduino Uno etc.

Keywords: IoT security · Kernel tracing · Embedded device security

1 Introduction

Internet of Things (IoT) is an emerging domain used widely in deploying embedded systems in the real world. This deployment provides fascinating opportunities for future applications. IoT devices include home alarm systems, medical devices, cameras, cell phones, vehicles, smart cities, buildings, running shoes, refrigerators and ovens etc. Such deployments also have their own short comings [19]. IoT nodes capture valuable data and provide smart services that may be of value to adversaries. Adversaries may tamper with the device, insert malicious payload or intervene overall operation of the IoT device. Unfortunately, embedded system vendors are careless about the software security because of which adversaries often cause damage to the IoT device. The common vulnerabilities present inside a code sample of an IoT device are buffer overflow, array out of bound, illegal pointer access etc. These vulnerabilities allow an adversary to execute code reuse attack and tamper with the devices, intervene their communication channels, or clone the devices, to instrument the data gathering and overall operation for their own interest [4]. Estimation of secure state in a device is the dynamic problem of estimating the state of a system and detect an adversary's attack. Secure state estimation problem is basically focused on brute force attack, but this type of algorithm can terminate in polynomial time. In this paper, we proposed a novel algorithm that uses an attack detection module based approach to identify these attacks while a program is executing continuously.

Existing works [1,9,14] on securing embedded devices are focused on system settings and injecting run-time checks into the binary. In their work, they assumed that, an adversary can tamper the control flow [14], hijack the control from vulnerable function [1] and perform code reuse attack [17], DoS attack [2], memory corruption attack [8], etc. These settings and assumptions primarily focus on the following points:

- How to verify control flow with low overhead in presence of control flow hijack attack [14,17]?
- Mount perfect buffer overflow attack, identify their conditions of existence, analyse the effect of code reuse attack and there mitigation policies [3,10].

Present solutions prevent control flow integrity and apply these attack-defense matrices in a special type architecture i.e. x86, but most of the IoT devices use a heterogeneous architecture like ARM and Intel. In this paper, we present an architecture independent solution, which is suitable for all the available architectures of IoT nodes.

```
ppoll([{fd=3, events=POLLIN}], 1, NULL, [], 8) = 1
([{fd=3, revents=POLLIN}])
recvmsg(3, {msg_name(0)=NULL, msg_iov(1)=[{"L\2\0\0
\0\0\0\0\10\0\0\0renderer\r\0\0\0\16\0\0\0/pro"...,
8192}], msg_controllen=40, [{cmsg_len=40, cmsg_level=
SOL_SOCKET, cmsg_type=SCM_RIGHTS, [17, 18, 19, 20, 21,
22]}], msg_flags=0}, 0) = 592
pipe([23, 24]) = 0
clone(child_stack=0x7ffd0405ebb0, flags=CLONE_NEWPID|
SIGCHLD) = 2484
close(23)                                    = 0
close(17)                                    = 0
recvmsg(3, {msg_name(0)=NULL, msg_iov(1)=[{"\10\0\0\0\4\0
\3426\0\0", 8192}], msg_controllen=0, msg_flags=0}, 0)=12
write(24, "\3426\0\0", 4)                    = 4
close(24)                                    = 0
```

The above listing, shows system call traces of an IoT device showing a series of system calls executed on the IoT device. We evaluated the implications of intermittent attacks on IoT devices similar to Fig. 1 and estimated the system call execution probability. With the help of quantitative knowledge about the system call tracing and detection of an attack from kernel level, we propose a non-intrusive tracking of an attack in IoT environment such that resultant IoT devices are robust against such kind of attacks. The proposed method tracks all illegal executions and ensures the stability of the IoT device with minimal overhead, based on the software deployment irrespective of extra hardware support.

Rest of the paper is organized as follows: Sect. 2 shows the threat model used for this research. Section 3 contains the problem statement that is solved in this

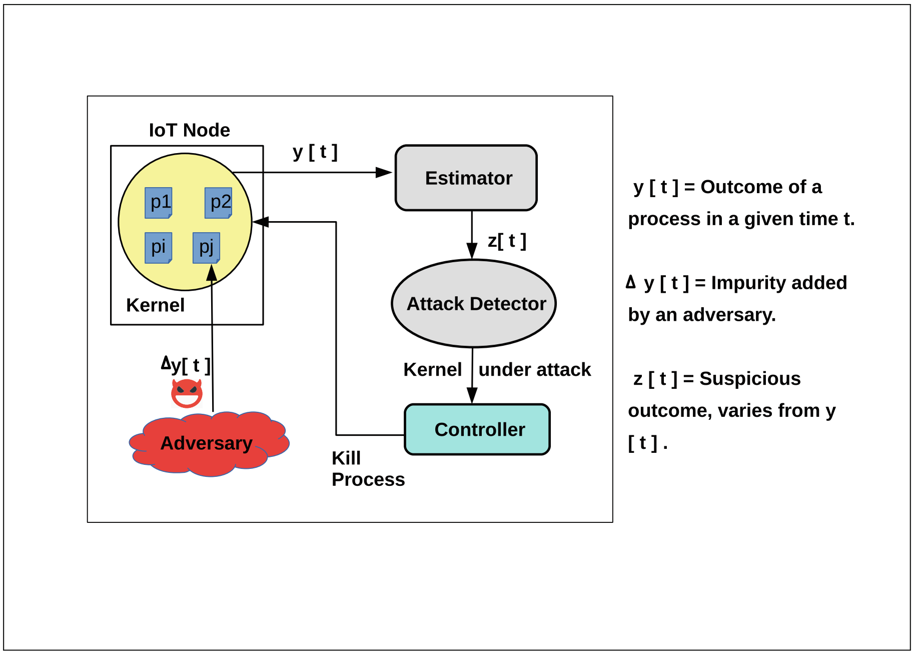
IoT Node
p1
p2
pi
pj
Kernel
y [t]
Estimator
z[t]
Attack Detector
Kernel under attack
Controller
Kill Process
Δy[t]
Adversary
y [t] = Outcome of a process in a given time t.
Δ y [t] = Impurity added by an adversary.
z [t] = Suspicious outcome, varies from y [t] .

Fig. 1. Attack detection model

research. Section 4 focuses on our proposed model. This section also discusses our mathematical model to identify an attack in IoT environments. Section 5 gives detailed information of the test-bed setup and functionality of each module deployed in this proposed approach. Section 6 discusses, how proposed model is suitable to detect an attack from the kernel level traces and also presents the result analysis. Section 7 discusses the related research and the contribution of other researchers to securing embedded devices. Section 8 concludes the work and provides direction for the future aspects of this work.

2 Threat Model

We used a hostile host threat model [5,13] to evaluate the proposed work. In this model, attacker can execute any program and control memory and execution environments. An attacker can also modify existing binary by using debugging tools.

Additionally, as part of the threat, an attacker is assumed to be capable of the following actions:

1. Collecting information about the system through some reverse engineering techniques.
2. Identifying the presence of already available system level defences such as ASLR, DEP, etc.

3. Injecting faulty data and mounting an attack from remote location to hijack an IoT node.

The primary objective of an adversary is to perform code tampering with the executable code and modify the functionality of the software. A hostile host model allows an user to perform reverse engineering methods to alter the existing code or add new functionalities into the code. The objective of our code tamper-proofing method is to protect the code tampering by non-intrusive monitoring. It traces attempts made by the adversary to tamper the protected binary.

The threat model assumes that the running program must have memory corruption vulnerabilities that allows an attacker to write random values to random memory locations including stack and heap and also leaks some information by arbitrary read primitive.

3 Problem Formulation

The objective of this paper is to build a kernel-based monitoring solution that supports a non-intrusive monitoring of running processes and threads at the operating system level. This monitoring method must trace all the activities carried out in an IoT device without affecting the running program. It would use a method that allows the kernel to register all events related to processes and threads. For example, Process creation, Process termination, Execution of a system call, etc. The most important thing that we consider, is that, the deployment exhibits high performance with low overhead.

We formulate our objective as a linear time invariant system problem. A time-invariant system does not directly relate to time, but the state of a system varies between two different time intervals on the basis of input and output of a function. In this work, we consider a system which has an output function $y(t)$, and an input function $x(t)$. Both functions are dependant on time and hold a relation like $y(t) = x(t + \delta)$, where δ is the difference between start and end time. Then, we can detect any illegal activity or an attack performed on an IoT device by analysing the input and the output functions.

4 Proposed Model

The objective of this paper is to detect adversary's activities by tracking the kernel activities in real time. We consider a discrete linear time-invariant model [2,9] to demonstrate our proposed problem. Let us assume,

$$x[t+1] = \alpha x[t] + u[t] \tag{1}$$

and the output is presented by,

$$y[t] = \beta x[t] \tag{2}$$

where, $x[t]$ is the matrix which contains system calls executed by each process in a given time t. $y[t]$ is the expected output or state of all the processes with respect to system calls within a given time t. α and β are constant values set by the administrator to calculate the next state. The $u[t]$ is the estimated update of individual process state. $u[t] = \gamma\,\hat{x}[t]$, where γ is the estimated count of a system call at time t in a specific process. The $\hat{x}[t]$ is the expected process state generated by the estimator using kernel output $y[t]$. The estimator function has a predefined threshold value θ. If the value of $u[t]$ is higher than θ, then, estimator function transfers $u[t]$ matrix to attack detector module which analyzes $y'[t]$ (received output instead of $y[t]$). If the attack detector module detects malicious activity in the kernel, then, it initiates a signal to the controller and the controller can kill the malicious process. Figure 1 shows the proposed approach. This model uses the hostile host attack model. This model assumes that an adversary has control over execution environments and he/she can execute malicious program to eavesdrop on the kernel activity and system calls, which are responsible to produce $y[t]$. The adversary can inject an impurity Δy into any running process by identifying vulnerability [9]. Due to this malicious attack, the estimated output value $\hat{x}[t]'$ is different from the predicted value $\hat{x}[t]$. $z[t] = (\hat{x}[t]' - \hat{x}[t]) > \theta$, is a sufficient condition to raise an alarm. The controller function determines the process id, which is malicious, and sends a kill signal to the kernel to terminate the process.

5 Experimental Set-Up

Figure 2 shows the test-bed setup. Raspberry Pi 3, and Intel Galileo Gen 2, devices have been used in this experiment. The “strace -p” command is used to collect real time logs of process executions. All the logs are transferred to server by using socket programming. A daemon process running in the back-end continuously collects logs and transfers them to the server. A centralized server is maintained for collecting all the logs and processing the data for extracting useful information. The experiment can be classified into the following phases.

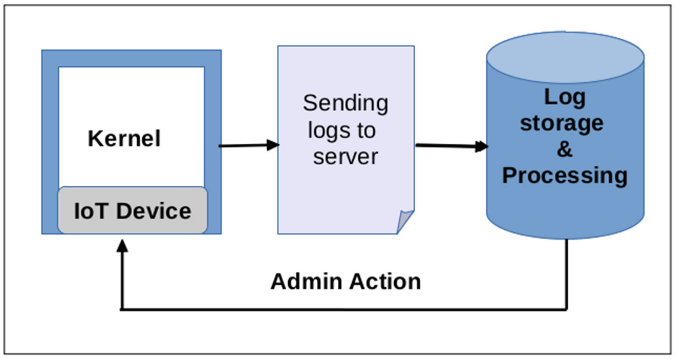

Fig. 2. Test-bed

5.1 Kernel Trace

This phase captures process events like process creation (*PROC_EVENT_FORK*), execution of system call (*PROC_EVENT_EXEC*), change in user id (*PROC_EVENT_UID*), process termination (*PROC_EVENT_EXIT*), thread creation and thread exit. The kernel level traces allow event registration related to processes and threads. It also generates the log of each process id (pid). In this module, the kernel traces each newly created process. Once the kernel tracer receives pid, it collects system calls executed by pid and sends to the server. The server has multiple monitoring processes to monitor each syscall trace generated by the process.

5.2 Estimator

The IoT device can execute predefined tasks like read, pre-process and write sensor values. This phase processes prior knowledge about system calls executed by specific process in a given time interval, i.e., $x[t+1] = \alpha\, x[t] + u[t]$ and expected output $y[t]$. The threshold value θ is used to differentiate between attacks and system noise. Now we define, τ, such that, $\theta = \{x[t] + \tau\}$, i.e., in a given time t, if $count(x)$, i.e., total number of system call count in a given time interval, crosses θ, then this can be labelled as an attack. If $x[t{+}1] - x'[t{+}1] > 0$, where $x[t+1]$ is expected value and $[x+1]'$ is real value provided by kernel, then there is a chance of an attack. Then the value of $x'[t+1]$ is sent to attack detector module to verifying the attack.

5.3 Attack Detector

Once the quantifier threshold θ is decided, the attack detector revisits log file of suspicious pid and evaluates all system calls executed by the process within the given time. Attack detection can be computed as follows:

$x_1 = s[t+1] - s[t];$
$x_2 = s[t+2] - s[t+1];$
...
$x_n = s[t+n] - s[t+n-1];$
$i.e., \forall x_i - (x_i - 1) \approx \theta \underline{+} \tau$

If there is any sudden increase in the existing system call count or any suspicious system call, such as, `memcpy`, `syscall_handler`, `ptrace`, etc. getting detected by the attack detector, then $\forall a \,\epsilon\, x_i \, and \, x_i \cap a \neq x_{i-1}$, "$a$" can be labelled as unregistered. An alert is raised in accordance to the list given below:

```
si_signo=SIGCHLD,  si_code=CLD_TRAPPED,
si_pid=1709,  si_uid=1000,   si_stime=1
si_status=SIGTRAP,  si_utime=4
```

In the above listing, "pid 1709" tries to debug ptrace signal of another process. The controller module issues a kill signal to the processor for the suspicious pid.

6 Result Analysis

Figure 3 shows an analysis of system call traces for a process. It indicates run time execution and call counts, which are monitored by the proposed model.

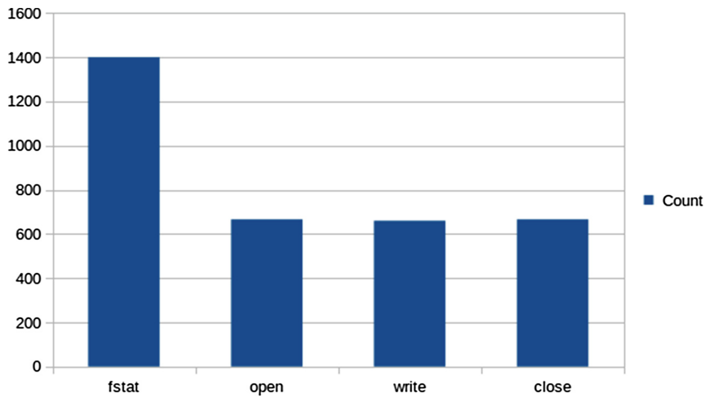

Fig. 3. Syscall traces of a process

6.1 Attack Matrix

- **ROP (Return Oriented Programming) Attack:** The binary running on an IoT device may contain a vulnerable function that can be attacked by an adversary. We exploit this binary using buffer overflow by uploading malicious payload. This malicious payload is stored in the stack. It tries to execute another program with the help of "exec" system call. This attack also uses system() function to invoke another program. The common system calls used by this attack are system(), execve() etc. The return address of a function subvert the control flow and launch a new shell to replace the current running program.
- **JOP (Jump Oriented Programming) Attack:** This attack uses functions which are vulnerable to buffer overflow such as memcpy(), strcpy(), sprintf(), strcat(), sscanf() and fscanf() etc. The JOP attack uses setjmp() and longjmp() functions to mount the attack. In JOP, the adversary used "jmp_buf" structure available in "setjmp.h" and calls setjmp(), which keep the current IP value and longjmp() returns control flow back to setjmp(). Our adversarial program overwrites this "jmp_buf" buffer and then calls longjmp() to transfer the control flow to another location. This attack executes new system call into the kernel and bypasses some existing system calls.
- **Process Tracing:** Reverse engineering tools like "ptrace" is used to trace the running processes. Adversary can control or analyse and manipulate the internal state of user program.

Our attack module attaches malicious binary to the running process. The attack module performs manipulation of file descriptors, memory, and register values. This module observes and intercepts system calls and their results. It can also manipulate the signal handlers and can also receive and send signals.

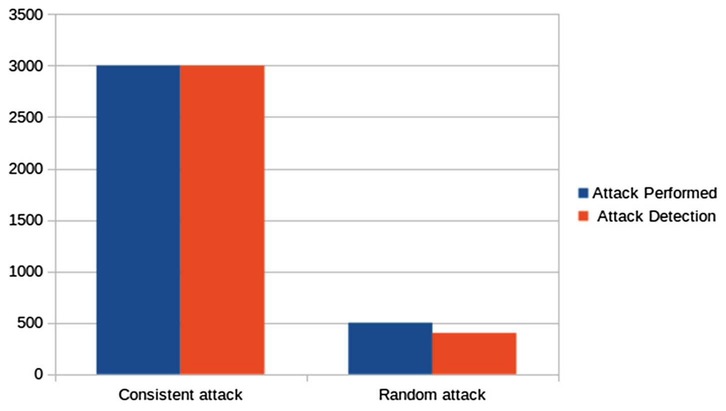

Fig. 4. Attack Evaluation

Figure 4 depicted the attack detection by our proposed approach, we execute attacks in following two different approaches:

- Consistent attack [15]: Consistent attack is a type of bias attack, it continuously attacks the target program and tries to modify binary. We ran a program which continuously execute "ptrace" and collect debugging information. This type of attack is identified by the work done in this paper with 100% success rate.
- Random attack [15]: In this attack, attacker randomly tries to modify or mount an attack. We applied the same program for executing attack, but the program repeat itself in random time interval. This attack is detected only when an attacker's program crosses the binary count threshold value. Our work, in this detects random attacks with 80% success rate.

6.2 Attack Countermeasures

The experiment environment executes only one user program "P" in the kernel for monitoring, which reads sensor value and after pre-processing sends it to the server. With the kernel output for a sampling period "T", such that the kernel state $x[t+1]$ during the $(t+1)^{th}$ sampling period as given in Eq. 1. Let us assume S = {y=x}, i.e., initial state as shown in Eq. 2 and there is no attack on

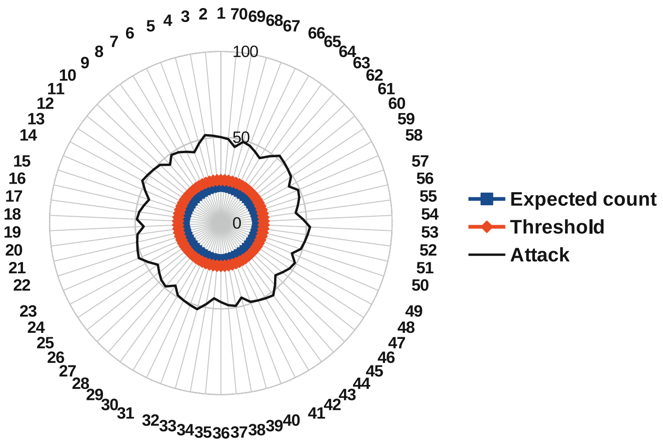

Fig. 5. Attack detection using proposed method

the kernel. In this case, we can easily estimate the updated value u[t]. Updated value of a single program is represented as {x[t+1] = Ax + u; P} such that { [x = Ax + u; P]^n} represents nth state of user program P and "A" is an execution count of "x".

Figure 5 shows a scenario of an attack detection using system call count within the time interval. Let the signal (system call count) for a given time interval is r, so x[t] = r is the acceptable threshold. But in some cases we have $x[t] = r + \tau$ for some $\tau \epsilon \theta$, and $x[t] \epsilon S$, where S is kernel space with variation τ having expected value r. Hence, an adversary is allowed to inject maximum τ values in each time interval. But there exists no mathematical model to estimate τ for an adversary to inject malicious payload.

Table 1. Process id and system call count

Process Id	System call	Count
1898	recvmsg	33980
2220	gettid	32546
13275	ptrace	2386
13296	ptrace	34862
13317	rt_sigprocmask	360

Table 1 shows the system call count made by an individual process. When an adversary executes malicious code, the kernel tracer identifies new "pid" which uses "exec()" or "system()" calls. This unregistered "pid" and system calls alert the attack detector module and controller module and sends a kill signal to the kernel.

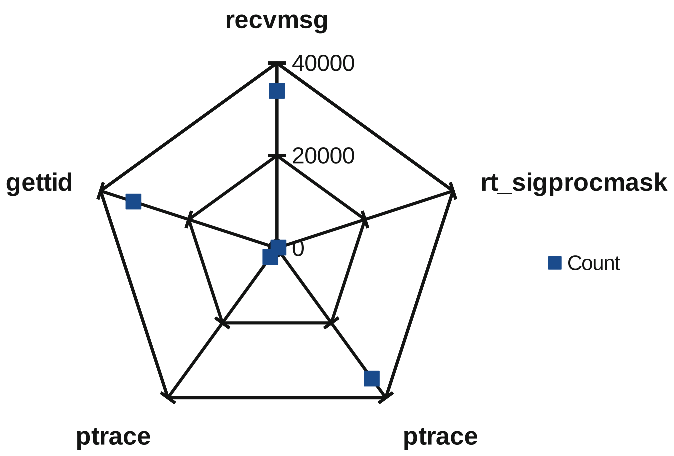

Fig. 6. Top 5 system call trace

Figure 6 exhibits traces of top 5 system calls, here system calls "recvmsg", "gettid", "rt_sigprocmask", "ptrace" are used by kernel trace program. But there is one more "ptrace" command executed by unregistered pid. This system call increases the count of expected "ptrace" system call count and attack-detector module raises an alarm for the controller.

6.3 Performance Overhead

Figure 7 shows the memory graph of the proposed method. All experiments were performed on Raspberry Pi 3 and Intel Galileo Gen2 IoT boards. Here the target program executes constantly and is not interrupted by the kernel trace program. The kernel monitor program collects information of each process and transfers it to the server. This kernel program runs like a daemon process and does not generate overhead for a running process.

7 Related Work

Nagarkatte et al. [12] proposed a hardware-based solution. They generated a meta-data of each pointer used in the program binary. This meta-data contained the base addresses for a pointer. At the time of execution the pointer memory range was verified by base address and the maximum allowable bound. The pointer based method had some loopholes, such as, more than one pointer variable reference a single memory location. Each pointer had its own memory location and meta-data which increase performance overheads and make programming more complex. Tsoutsos et al. [18] discussed various memory safety issues with respect to embedded devices, and described different attacks especially ROP that can hijack the control flow. They also suggested some attack

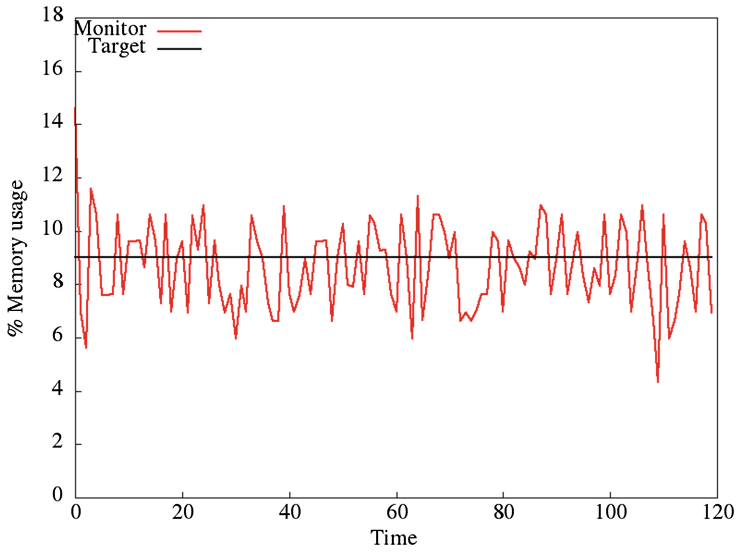

Fig. 7. Performance overhead

avoidance method like buffer bounds checking, instruction set randomization, and inline reference monitors etc. Krishnakumar et al. [8] developed a tool “Gandalf” which is the software based hardware extension to prevent memory corruption attack for the openRISC processor. Their solution depends on system’s hardware and attacks are determined by the hardware at runtime. Bresch et al. [1] presented a hardware based solution to prevent ROP attacks. The author has focused on micro-controller based devices to prevent code reuse attacks by modifying return addresses. Isenberg et al. [6] proposed a method for software verification against AC hardware. They performed array bound checking within the loop to avoid buffer overflow attacks. Shoukry et al. [16] proposed a solution for secure state estimation problem. In that solution they developed a simulation based attack on sensors in a linear dynamical systems with the presence of noise. Mo et al. [11] proposed a method to analyzed the effect of replay attacks on a tiny systems. They executed replay attacks to hijack the control flow of an embedded device. In that work, they injected an external input to hijack the sensors and record their readings. After a certain interval the attacks are repeated. They discovered that, for some systems, the classical estimation, control, failure detection strategies are not resilient to prevent replay attacks. They proposed a discrete time invariant model to discover the replay attacks. Jovanov et al. [7] used secure state estimator to prevent network-based attacks. They also proposed a sporadic data integrity enforcement method to prevent message authentication from stealthy attacks.

Most of the related work, focused on detecting the malicious attacks instead of taking action against these attacks. We proposes the complete solution, i.e attack detection and kill the malicious process. Our solution is independent from hardware and the architecture of IoT board.

8 Conclusion

The experiments carried out in this paper are mainly used for diagnostic and debugging kernel space utilities for IoT nodes. The proposed solution monitors any adversarial attack by monitoring the kernel's traces, which include system calls, signal deliveries, and change of process states. In future, we intend to improve kernel tracing for multi-process non-linear systems.

Acknowledgement. This work was supported by Ministry of Electronics and Information Technology (MeitY), Govt. of India and Netherlands Organization for Scientific research (NWO), Netherlands under grant number: 13(1)/2015-CC&BT.

References

1. Bresch, C., Hély, D., Papadimitriou, A., Michelet-Gignoux, A., Amato, L., Meyer, T.: Stack redundancy to thwart return oriented programming in embedded systems. IEEE Embed. Syst. Lett. **10**(3), 87–90 (2018)
2. Ghosh, S.K., Dey, S., Mukhopadhyay, D.: Performance, security trade-offs in secure control. IEEE Embed. Syst. Lett. **11**, 102–105 (2018)
3. Habibi, J., Panicker, A., Gupta, A., Bertino, E.: DisARM: mitigating buffer overflow attacks on embedded devices. Network and System Security. LNCS, vol. 9408, pp. 112–129. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25645-0_8
4. Ho, J.-W.: Efficient and robust detection of code-reuse attacks through probabilistic packet inspection in industrial iot devices. IEEE Access **6**, 54343–54354 (2018)
5. Hota, C., Shrivastava, R.K., Shipra, S.: Tamper-resistant code using optimal ROP gadgets for IoT devices. In: 2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC), pp. 570–575. IEEE (2017)
6. Isenberg, T., Jakobs, M.-C., Pauck, F., Wehrheim, H.: Validity of software verification results on approximate hardware. IEEE Embed. Syst. Lett. **10**(1), 22–25 (2017)
7. Jovanov, I., Pajic, M.: Sporadic data integrity for secure state estimation. In: 2017 IEEE 56th Annual Conference on Decision and Control (CDC), pp. 163–169. IEEE (2017)
8. Krishnakumar, G., Slpsk, P., Vairam, P.K., Rebeiro, C., Veezhinathan, K.L.: Gandalf: a fine-grained hardware-software co-design for preventing memory attacks. IEEE Embed. Syst. Lett. **10**(3), 83–86 (2018)
9. Li, Y., Shi, D., Chen, T.: A stackelberg-game analysis, false data injection attacks on networked control systems. IEEE Trans. Autom. Control **63**, 3503–3509 (2018)
10. Liu, J., Sun, W.: Smart attacks against intelligent wearables in people-centric Internet of Things. IEEE Commun. Mag. **54**(12), 44–49 (2016)
11. Mo, Y., Sinopoli, B.: Secure control against replay attacks. In: 2009 47th annual Allerton Conference on Communication, Control, and Computing (Allerton), pp. 911–918. IEEE (2009)

12. Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: WatchdogLite: hardware-accelerated compiler-based pointer checking. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, p. 175. ACM (2014)
13. Nickerson, J.R., Chow, S.T., Johnson, H.J.: Tamper resistant software: extending trust into a hostile environment. In: Proceedings of the 2001 Workshop on Multimedia and Security: New Challenges, pp. 64–67. ACM (2001)
14. Nyman, T., Ekberg, J.-E., Davi, L., Asokan, N.: CFI CaRE: hardware-supported call and return enforcement for commercial microcontrollers. In: Dacier, M., Bailey, M., Polychronakis, M., Antonakakis, M. (eds.) RAID 2017. LNCS, vol. 10453, pp. 259–284. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66332-6_12
15. Park, J., Ivanov, R., Weimer, J., Pajic, M., Lee, I.: Sensor attack detection in the presence of transient faults. In: Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems, pp. 1–10. ACM (2015)
16. Shoukry, Y., Nuzzo, P., Puggelli, A., Sangiovanni-Vincentelli, A.L., Seshia, S.A., Tabuada, P.: Secure state estimation for cyber-physical systems under sensor attacks: a satisfiability modulo theory approach. IEEE Trans. Autom. Control **62**(10), 4917–4932 (2017)
17. Shrivastava, R., Hota, C., Shrivastava, P.: Protection against code exploitation using ROP and check-summing in IoT environment. In: 2017 5th International Conference on Information and Communication Technology (ICoIC7), pp. 1–6. IEEE (2017)
18. Tsoutsos, N.G., Maniatakos, M.: Anatomy of memory corruption attacks and mitigations in embedded systems. IEEE Embed. Syst. Lett. **10**(3), 95–98 (2018)
19. Zhao, K., Ge, L.: A survey on the internet of things security. In: 2013 Ninth International Conference on Computational Intelligence and Security, pp. 663–667. IEEE (2013)