2014 IEEE Security and Privacy Workshops

Using Existing Hardware Services for Malware Detection.

Sarat Kompalli, Intel Corporation

Abstract—
The paper is divided into two sections. First, we describe our experiments in using hardware-based metrics such as those collected by the BPU and MMU for detection of malware activity at runtime. Second, we sketch a defense-in-depth security model that combines such detection with hardware-aided proof-carrying code and input validation.

***Keywords*—malware, security in hardware, data security**

I. Experimental Setup and Results

This section describes the experimental setup of malware activity detection based on Branch Prediction analysis and memory mapping.

A. Branch Prediction analysis-based filters

1) OS, platforms, and applications used

In this study, we use the modified Win32/Renos malware for contaminating our systems [1]. The original malware has been further modified to add buffer overruns, for experimental use by our internal teams. Vtune [2] is used to monitor the Branch Prediction Unit (BPU). For this experiment we use Intel HSW CPU T245 up to 2.67GHz. For this experiment we developed a special C++ checker (Active and Passive safety net, APSN). APSN acts as a wrapper around Vtune. The primary function of APSN is to automate data analysis. APSN is pre-loaded with a cache of BPU data across multiple flavors of applications, drivers, software etc. APSN is re-loaded with new BPU data before making any changes to the hardware or software stack.

2) Training and calculating thresholds

We run the systems with baseline applications and use Vtune to report BPU data. All application, driver, and system information is kept constant throughout the training. APSN performs statistical analysis on the BPU data and determines a tripping point for the clean systems. The tripping point is the maximum percentage of errors made by the BPU over the entire dataset. Figure 1 reflects BPU data from systems that are unaffected with malware, aka clean systems. Our clean systems are running with standard v2.3.4 BIOS, firmware, Windows Vista and have been scanned for known malware [3].

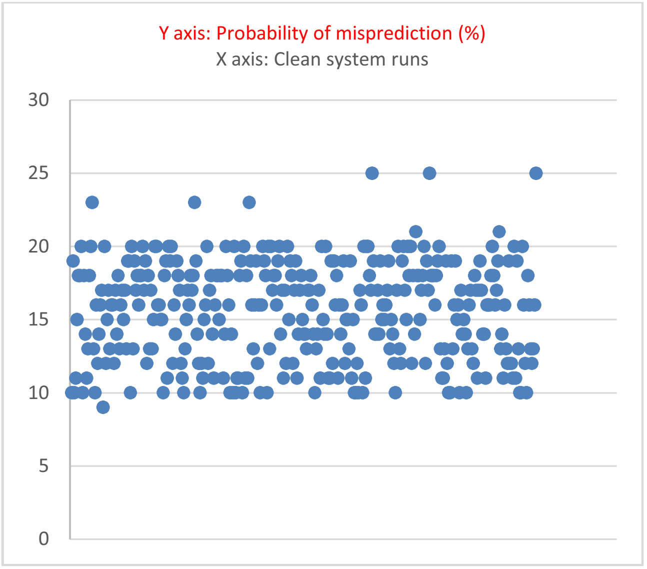

Figure 1: BPU data from clean system runs

For the list of applications used in our experiments, APSN calculated a max value of 34% (10% guard-band to minimize false failures). In our further field experiment, any process generating BPU value over the max value of 34% is determined as malware by APSN.

3) Collecting data on malware-affected systems

In this part of the experiment, we use “dirty systems”, which are the clean systems above affected with Win32/Renos. The usage patterns are kept similar to the patterns used in the “clean systems”. We re-collect the BPU data by executing similar load experiments. Figure 2 below reflects the data from the dirty system runs.

DOI 10.1109/SPW.2014.49

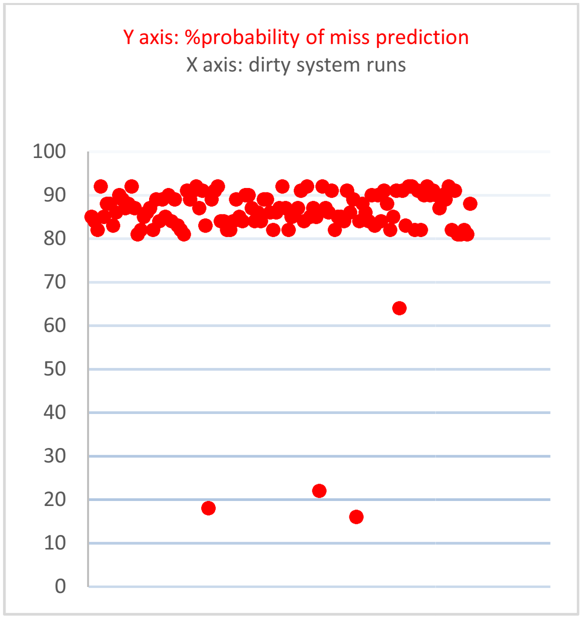

Figure 2: BPU data from dirty systems (malware affected)

4) Overlaying data from clean systems onto dirty systems:

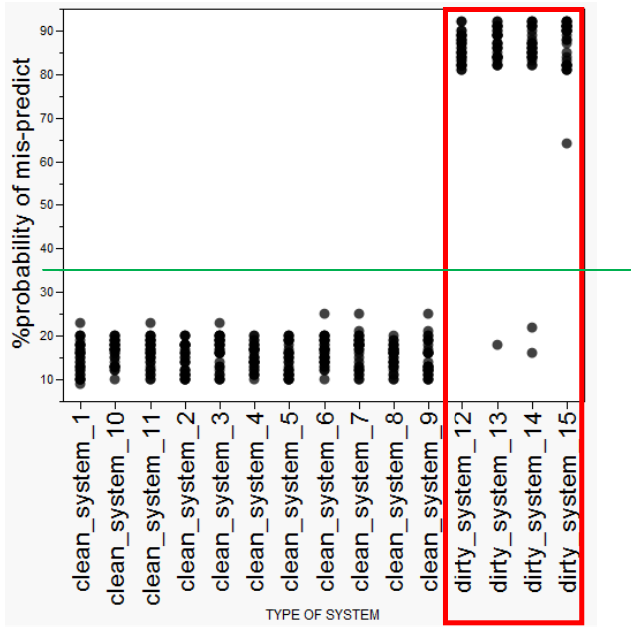

Figure 3: Overlay of data from malware affected and non-affected systems

As we can see from the above chart (figure 3), the branch prediction miss rates are below threshold for a clean system, i.e. the Branch Prediction Unit (BPU) is very accurate in predicting the next branches. However, in the dirty systems (infected with malware), the BPU produces high rate of prediction misses. APSN system is designed to generate a malware detect signal when a process goes over the threshold limit.

High level pseudo-code of the APSN filtering:

```
if(BPUdata >= threshold (max%probability_to_miss_predict)
    {
        Malware_detect_signal = 1'b1
    }
```

The above data indicates that BPU readings can potentially be used to detect malware. However, the outlier data points are concerning. Of the Y number of runs on the dirty systems, X went undetected by the threshold, forming the outliers shown in Figure 4, red box. These outliers are cases where the malware affected system generated BPU data under the 30% threshold limit. We believe that using a BPU threshold alone as the indicator is not sufficient.

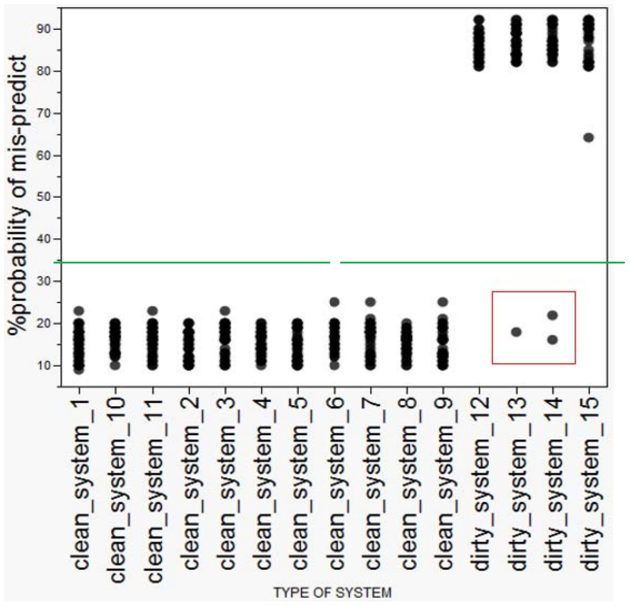

Figure 4: Highlighting outliers that APSN does not catch

In the subsequent section we describe a system around BPU, which will help detect the outliers highlighted in figure 4.

B. Memory mapping based filter

Memory monitoring has been used previously for rootkit detection [4]. We are applying these concepts towards malware detection. To the best of our knowledge, this is the first such attempt.

1) Runtime Memory spoofer: Dynamic Memory Monitoring Engine (DMME)

Our next checker is based on the principle of monitoring runtime memory allocation and usage [5]. Experimental setup: we first run many cycles on our clean system and build a behavioral model. The data is collected using special system validation tools using debug API's running under *red unlock* privileges. SoC/Processor companies lock their debug capabilities to protect their architectures. Semiconductor devices require appropriate authorization to use functions available during manufacturing, which it locks down before shipping the products. Specialized software and hardware is required to authenticate this level of unlock, commonly known *red-unlock*.

The DMME model measures 4 parameters for each application:
Memory locations (ML)*: the number of memory locations a specific application accesses.*
Number_of_reads + variance f(Rn)*: the number of reads done by the application plus a pre-calculated guard band to account for variations.*
Number_of_writes + variance f(Wn)*: the number of writes done by the application plus a pre-calculated guard band to account for variations.*
Max Memory size (MS)*: memory size that each of these applications is reserving or requesting*.

f(Rn), f(Wn), f(MS) are computed using controlled software runs. Figure 5 describes the major blocks and interaction of the memory spoofer. In our system, applications request memory transactions via the memory manager (MM). The main purpose of the MM is to act as a bridge between the application layer and the CPU. As its secondary role, the MM also feed-forwards the application type and its memory request data to the Dynamic Memory Monitoring Engine (DMME). The DMME tallies this constant feed of data to the pre-calculated values of f(Rn), f(Wn), F(MS) and checks to make sure the data from the MM is lower than values present in the behavioral model. If an application violates thresholds, the DMME produces a report for the APSN. The report contains application id and memory details. The APSN uses this signal to tag its application as high risk and requests BPU data to be re-scrutinized.

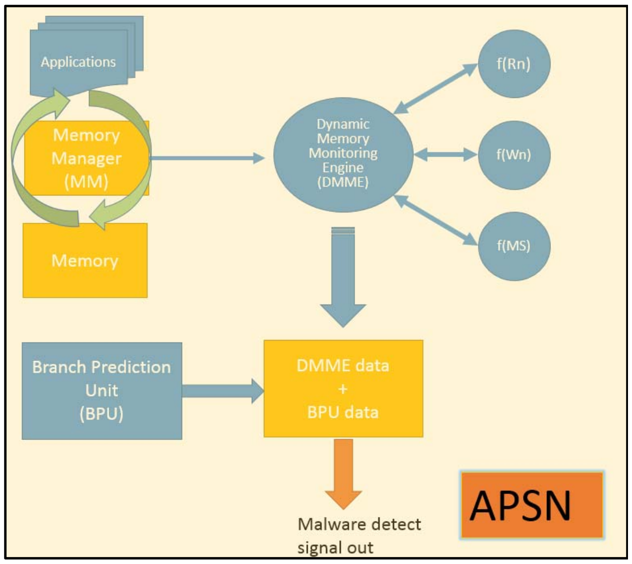
Applications
Memory Manager (MM)
Memory
Dynamic Memory Monitoring Engine (DMME)
f(Rn)
f(Wn)
f(MS)
Branch Prediction Unit (BPU)
DMME data + BPU data
Malware detect signal out
APSN

Figure 5: APSN system with DMME + BPU

We model the above blocks by re-running our clean_system and dirty_system and collect data on how many times our DMME generates a report for the APSN. Figure 6 shows an overlay of our BPU results with memory analysis data. It is evident from the graph that combining memory analysis data and BPU results would detect a large percentage of malware calls. There is a chance of obtaining some false positives. For example, in our case 1 run out of X was a false positive. However, there is a promise of being able to have a low false negative rate as opposed to using BPU alone.

🟥F : DMME generated the trigger.
🟩P : DMME did not generate the trigger.

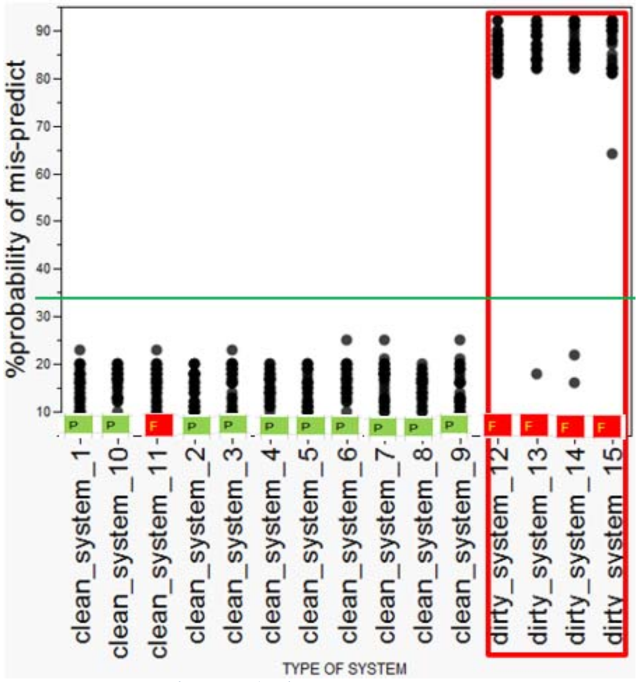

Figure 6: DMME data overlaid on BPU

This approach of using BPU and memory mapping data be reliably used for malware detection.

High level pseudo-code APSN+DMME:

```
if (BPUdata >= theshold (max%probability_to_miss_predict)
    {
        andif (DMME_MM_triggered == 1'b1) {
            malware_detect_signal = 1'b1
        }
        elseif (DMME_MM_triggered == 1'b0) {
            malware_detect_signal = 1'b0
        }
    }
```

Although our study provides a compelling solution for malware detection, a few points are relevant before closing the discussion.

1. Malware is evolving. The malware used in our study was very potent and created a noticeable difference between clean and dirty systems. It is likely that malware loaded applications can be developed to have very subtle variations in memory and BPU usage. It is necessary to continue further research on the implications of many degrees of malware on BPU data.

2. Interpreted languages are challenging. Though overlapping memory spoofing data over BPU creates a very clean filter for malware detection, more data collection needs to be done using applications developed on interpreted languages (like Java). Interpreted languages have a tendency to follow random patterns of memory fetches and can throw off our model behavior.

The above reasons make a case for us to explore *defense in depth* and understand the benefits of combining malware detection with *defense in depth*. In the following section of our paper, we propose a hardware based eco-system built around our APSN system.

II. Security Modelling by Combining Multiple Solutions

A. Proof-carrying code

Proof-carrying code, or simply PCC is a mechanism that allows *code consumer* to check the validity of the code without using cryptographic techniques [6]. PCC increases security, with very little or no performance overhead during run-time [7]. PCC allows the *code consumer* to define a safety policy and then validate pre-run time that this policy has been satisfied [8]. Adding proof-carry checkers to our APSN gives us additional security against malware coded in interpreted languages. The idea is to have software vendors attest their software/applications with embedded signatures. Also, any new software, driver, applications need to pre-populate the APSN with its BPU data predictions [9].

B. Hardware-based input validation

Several attacks can be run against a web application, inserting crafted data — often, too much at once — which can confuse, crash, or make the web application divulge too much information to the attacker. Buffer overflow attacks are the best examples to prove the importance of input validation. In this last section of our paper we propose a hardware-based input validation technique, the idea is similar to a few academia proposals on input parsing [10].

Network intrusion detection systems (NIDS) [11] are an efficient mechanism for passively examining network traffic to determine whether a packet contains an attack payload or not. Our proposal is to add input parsing as a by-product of this packet examination.

Traditional NIDS systems are designed to match a specific known string. One or more string matches are combined into a single rule used to define a signature. However, static and loose signatures can increase the number of false positives.

The answer to static pattern matching has been to use hardware based regular expressions (B-FSM) [12]. A single regular expression can cover a large number of static matches, and thus regular expressions have become essential for combining many types of detected patterns. Our approach is to simplify the input to well-defined chunks of data that is linked together with known constraints. For example, when a user inputs login details, hardware will perform the following steps. (1) Parse the input, and tabulate the data into a simple format. (2) Run specific pattern matching on each field of the tabulated data, in this case *[a-z][0-9][len_max:12].*

This tabulation of inputs into known fields will remove any ambiguity before the code is executed, thus reducing the risk of attack. Unfortunately, today's regular expression matching schemes that are implemented in software pose a very high performance overhead. One solution for bypassing the performance overhead is to implement the regex matching in hardware.

A few previous solutions have used hardware to perform this complex computation [13]. Some previous solutions were to implement the regex matching logic on programmable devices like FPGAs. Though they are efficient, they have a few drawbacks.

1. They are complex to modify and program.
2. FPGA process size is different from current mainstream process (14nm, or 22nm). This process difference forces the regex logic and main CPU to be on different pieces of silicon, thus leading to complicated power flows and clock management, eventually adding to the cost of manufacturing and testing.

To overcome the above drawbacks, our solution is to move the regex matching logic inside the CPU/SoC silicon. We propose developing an input parsing and regex matching (IPRM) module inside the CPU/SoC. This module would be made up of an input parser, and a malware regex matcher.

Input Parser (IP) has two main duties. The primary duty is authentication of the PCC. The IP needs to authenticate the proof of the code, by running it against the LF theorem checker [14]. Once the proof is authenticated, the IP parses the data and tabulates it into pre-determined fields for a given application.

Malware Regex Matcher (MRM) will monitor the parsed code for any malware strings. The MRM can be run in two modes. In mode 1, it analyzes the network packets and displays them on the console. In mode 2 (default) the MRM will analyze the packets and check them against a pre-defined regex defined by the hardware. Broadly, it is responsible for acting like a network intrusion detection system [15].

Incoming raw data is first processed by the IP. If the PCC and tabulation are successful, the IP passes the processed information to the MRM. The MRM will then analyze the packets for any malware. Any failures in either MRM or the IP add the application/software to the quarantine report. The system communicates these reports back to application vendors. IPRM is designed so that MRM and IP order can be interchanged in the pipeline.

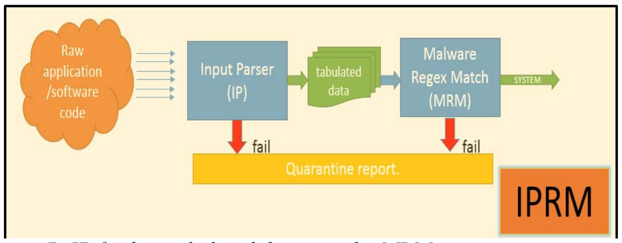

Figure 7: IP feeding tabulated data into the MRM

III. Summary

Our paper presents two important layers of security, one during runtime at the system level via our APSN, and second layer of security via our IPRM. We prove the benefits of overlaying the BPU data over memory mapping data from the DMME and argue that using this combination provides us with a viable solution for detecting malware. We further enhance the level of security by using efficient support of the state-of-the-art regex and input handling implemented in hardware. We show the benefits of integrating the IPRM into SoC and main CPU silicon.

IV. Future Work

Our study in this paper was performed under very tight conditions on high-speed performance machines using a single flavor of malware. The BPU and DMME data needs to be characterized in-depth by using many different flavors of malware. We foresee changes to the thresholds as more and more flavors of malware and system combinations are introduced, and as data is analyzed. We also recognize the need for future work in the area of performance analysis around LF proof checks implemented in hardware and side-channel attacks using BPU data [16]. DMME currently is implemented at a restricted privilege level, this puts a lot of constraints on sharing performance-overhead and implementation details with external entities. We need to plan the abstraction of the API's needed for implementing DMME to a user-level privilege.

V. Related work

Some previous work has been performed in the area of using performance metrics for detecting and identifying security attacks [17] [18], however they fail to study the performance counter variations introduced due to inherited hardware un-maskable interrupts and system flaws and how their system responds to legit processes.

Acknowledgment

The author would like to thank David Grawrock and Dr. Sergey Bratus for their help with the preparation of this research paper.

References

[1] "Win32/Renos," Microsoft Corp., [Online]. Available: http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Win32%2fRenos. [Accessed 21 March 2014].

[2] "Intel® 64 and IA-32 Architectures Optimization Reference Manual," Intel Corporation, 2012.

[3] "McAfee Labs Threats Report," Intel Corporation, [Online]. Available: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2013.pdf. [Accessed March 13 2014].

[4] R. Junghwan, R. Ryan and D. X, "Defeating Dynamic Data Kernel Rootkit Attacks via VMM-based Guest-Transparent Monitoring," in Availability, Reliability and Security International Conference, 2009.

[5] "Software Concerns of Implementing a Resident Flash Disk," Intel Corporation, 1995.

[6] "Proof-carrying code - Wikipedia," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Proof-carrying_code. [Accessed 21 March 2014].

[7] The University of Texas at Dallas, [Online]. Available: http://www.utdallas.edu/~hamlen/Papers/necula96.pdf. [Accessed 31 March 2014].

[8] Carnegie Mellon University, [Online]. Available: http://users.ece.cmu.edu/~dbrumley/courses/18732-s12/slides/23-proof-carrying-code.pdf. [Accessed 31 March 2014].

[9] Z. Ramzan, V. Seshadri and C. Nachenberg, "Reputation-based security: An analysis of real world effectiveness," in VB, Geneva, 2009.

[10] B. Sergey and S. Sean W, Computational approach to trustworthy, programmer-friendly SoC architectures, unpublished, 2013.

[11] F. Yu, Y. Chen, T. Diao, T. V. Lakshman and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in Architecture for Networking and Communications Systems, 2006.

[12] "IBM Research," IBM Corp, [Online]. Available: http://www.zurich.ibm.com/sys/accelerators/bfsm.html. [Accessed 21 March 2014].

[13] v. L. Jan, R. Jon, A. Kubilay and H. Christoph, "Hardware-Accelerated Regular Expression Matching at Multiple Tens of Gb/s," in IEEE INFOCOM, 2012.

[14] "PRL Seminar," Cornell University, [Online]. Available: http://www.cs.cornell.edu/Nuprl/PRLSeminar/PRLSeminar01_02/Nogin/PRLseminar7b.pdf. [Accessed 1 January 2014].

[15] J. v. Lunteren, "High-performance pattern-matching for intrusion detection," in IEEE INFOCOM, 2006.

[16] A. Onur and S. Jean-Pierre, "Predicting Secret Keys via Branch Prediction," in The Cryptographers' Track at the RSA Conference, San Francisco, 2007.

[17] Y. Liwei, X. Weichao, C. Haibo and Z. Binyu, "Security Breaches as PMU Deviation: Detecting and Identifying Security Attacks Using Performance Counters," in Second Asia-Pacific Workshop, 2011.

[18] W. Georg, sysscan.org, [Online]. Available: http://syscan.org/index.php/download/get/3c6891f2e90e661ea23224cd8f419262/SyScan2013_DAY1_SPEAKER05_Georg_WIcherski_Taming_ROP_ON_SANDY_BRIDGE_syscan.zip. [Accessed 31 March 2014].