POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Interruptible Remote Attestation via Performance Counters

Laurea Magistrale in Computer Engineering - Ingegneria Informatica

Author: Davide Li Calsi

Advisor: Prof. Vittorio Zaccaria

Academic year: 2022-2023

---

1. Introduction

Low-power Microcontrollers (MCU) are a set of devices characterized by reduced cost, scarcity of computational power, and little memory. They also lack common hardware protections such as a Memory Management Unit. Despite these limitations, MCUs are widespread in several applicative scenarios (such as Smart Objects, Healthcare Devices, and Smart Sensors), making their security is a mandatory aspect. However their constraints drastically increase the difficulty of this task. In this struggle, Remote Attestation (RA) [2] is a powerful ally. It is a *challenge-and-response* protocol allowing an external entity, the **Verifier**, to verify the integrity of a target device, the **Prover** (in our setting, the MCU whose security is at stake). RA introduces has an inherent limitation: interrupts must be disabled during its execution. That is meant to guarantee the Attestation's atomicity, and ensures that no agent modifies the MCU's memory to escape detection. In fact some **roving malware** could use interrupts to move to previously attested memory, producing a misleading attestation response. Some solutions exist, yet they either provide modest detection probabilities or are simply unfeasible for low-end devices. Our work investigates innovative ways to perform interruptible RA on low-end MCUs.

2. State of The Art

2.1. Remote Attestation

Several types of RA exist. The most basic protocol consists in the Verifier sending a random challenge to the Prover, usually in the form of a **nonce** or a **timestamp**. The Verifier then expects the Prover to compute some evidence of its integrity using the challenge and its memory content. For simplicity, we restrict the computation to Program Memory only, as attesting Data Memory and Runtime Integrity is much more challenging. The evidence can be a simple checksum, a digest or an HMAC. The randomness of the challenge ensures that *replay attacks* are unfeasible. Upon reception the Verifier verifies the received evidence. This is possible because Program Memory is assumed to be quite static, with few known legitimate configurations. Therefore the Verifier can pre-compute evidences associated with benign configurations of the Program Memory. Several variations and classifications were formulated in the last decade. **Software Attestation** is run entirely via software, and is generally less secure than other forms. **Hardware Attestation** uses stronger Hardware protections to secure the Attestation Response. Finally **Hybrid Attestation** consists of a hardware-software codesign made of minimal hardware modifications that support

software-based checks. You can also differentiate based on what you attest. **Static Attestation** only attests Program Memory, while **Dynamic Attestation** also attests Runtime Integrity and Data Memory. Recent developments have also introduced **Swarm Attestation**, a protocol that attests a group of devices faster than point-to-point RA. Most protocols require interrupts to be disabled during the execution of the Attestation Routine (AR). This prevents attackers from exploiting interrupts to modify memory at runtime and escape detection. However interrupts are a key feature that ensures the responsiveness of the device. It is never a good idea to keep them disabled for too long. This is especially true when dealing with **real-time** applications. Motivated by these problems, researchers have investigated techniques for interruptible RA.

SMARM [4] addresses the issue by attesting memory in a pseudorandom order. The challenge also contains a seed s to determine a pseudorandom permutation of memory blocks. SMARM provides probabilistic guarantees depending on the attacker's knowledge of the system. The asymptotic detection probability, assuming malware fitting in s memory blocks and knowing only how much memory still needs to be attested, is $1 - e^{-s}$. For $s = 1$ this value is around 63%. Malware knowing which memory blocks were already attested has a detection probability of $\frac{1}{n}$ (n is the number of blocks). Furthermore interrupts are only partially allowed, since single block attestations still require atomicity.

Memory Locks [5] guarantee safe interrupts by locking memory. Locked memory is temporarily read-only. Because write operations are forbidden, roving malware is successfully stopped. The major shortcoming is that implementing memory locks require an underlying microkernel. The authors themselves state that microkernels of this complexity are unfeasible for low-end devices. This makes memory locks not applicable to this class of MCUs.

2.2. Performance Counters for Malware Detection

Malware Detection is a broad research field. Among all possible means, Performance Counters play a key role. They are hardware components natively present in many modern architectures. In practice, they comprise several hardware registers that are incremented whenever an event of interest occurs. Architectures often count low-evel events such as branches taken, CPI, cache hits/miss rates and so on. Their original purpose is different, either debugging or profiling, but they can also be exploited for security purposes [1, 9]. The majority of modern methods consist in two phases: an **offline phase**, in which you reproduce some attacks and collect a dataset of counters to train a Binary Classifier; an **online phase**, in which you read the available counters and feed them to the pretrained classifier. Across the years researchers have tested several classifiers, such as Decision Trees, SVM, K-NN and many others.

3. A new approach to Interruptible RA

We believe that it is possible to combine RA and Counters-based Malware Detection to allow interrupts during the AR's execution. In fact, even low-end devices possess Performance Counters for debugging. By activating them before the AR's beginning and reading them at its end, it is possible to determine whether roving malware has attempted to relocate or not. The implicit assumption is that malicious relocations significantly differ from legitimate applications, and Performance Counters allow to capture and measure this difference. The result is trivially a **counter vector** c in which each entry $c[i]$ stores the value of a counter monitoring a specific event. Like most existing techniques, our method requires a Binary Classifier to predict whether a target counter vector is the result of benign or malicious activity. We also investigated the effectiveness of adding high-level counters and feeding them to the classifier as well. While they add further complexity, due to their definition and secure management, we expect them to enhance the models' detection rates. We call these high-level counters **applicative**, while low-evel counters are labeled as **architectural**.

3.1. The target system's architecture

3.1.1 Uncontrollable Parameters

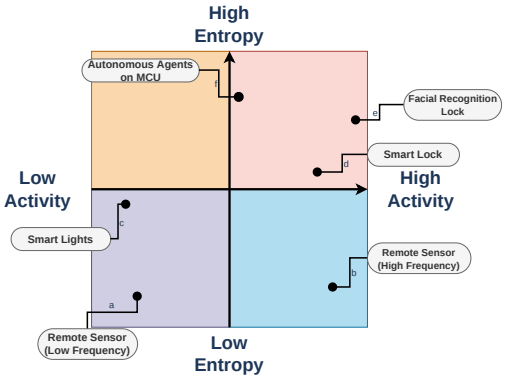
We start by analyzing the attacker and its characteristics. We do this by introducing a first class of parameters that determine the type of attack. We call them **uncontrollable** because they are out of our control. In fact, trivially we cannot choose in advance which type of attack our MCU is subject to. In this context, we focused on two such parameters: **malware size**, simply the size in KB; **malware type**, i.e. transient or self-relocating. **Transient malware** is a type of roving malware that self-overwrites with benign content in order to escape detection. Conversely, **self-relocating malware** moves to a new location and writes benign code to its previous location. Both parameters are relevant because they determine the number of operations required by a malicious relocation. It is reasonable to assume that more operations lead to a significant difference between malicious and benign counter vectors. Hence this should facilitate the classification task.

3.1.2 Controllable Parameters

Controllable parameters characterize the application running on the MCU. We call them controllable because they are under the system designer's control. However, it is important to keep in mind that they are *fixed*: once the stakeholders have chosen the type of application running on the MCU, we can hardly modify it. These parameters are particularly relevant because of their impact on benign counter vectors produced by the MCU. We defined two such parameters: **Entropy Level** (or simply Entropy), measuring the unpredictability of benign counter vectors; **Activity Level** (or simply Activity), measuring how large the counters are on average. Mathematically, Entropy is defined as the entry-wise Shannon Entropy of the counters vector

$$\mathbb{H}[c] = [\,\mathbb{H}(c[0]) \dots \mathbb{H}(c[n-1])\,]$$

Similarly, the Activity is defined as the expected value of counter vectors $\mathbb{E}[c]$. We argue that Entropy is a meaningful parameter because the more unpredictable the benign activity, the harder it is for a Classifier to learn it and distinguish from malicious relocations. Similarly, the higher the MCU's Activity, the harder it is to tell it apart from a large malware sample that is relocating.



Remote Sensors periodically sample and broadcast some quantities at fixed rates. At low sampling rates (a) they show low Activity and Entropy. Higher rates (b) do not affect Entropy, but the Activity skyrockets. Smart Lights (c) are simple remotely controlled lightbulbs. They typically do not undergo intense Activity, but interacting with a user makes them moderately unpredictable. Smart Locks (d) secure entrances, and can be triggered via smartphone. They require intense computation, resulting in high Activity. Entropy is also moderately high due to higher variability in the types of events. Facial Recognition Locks (e)[10] are Smart Locks using Machine Learning algorithms for Face Recognition. They sustain demanding computation, resulting in massive Activity levels and slightly higher Entropy. Finally MCUs implementing Autonomous Agents respond to external environments in a complex manner. This implies on average high Entropy and moderate-high Activity.

Figure 1: Example of real applications in the Activity-Entropy space

3.1.3 Hyperparameters

Both Controllable and Uncontrollable parameters concern the Prover. We now move on to discuss the major Verifier-side aspects. Our method only affects the Verifier by deploying a Binary Classifier on it. We therefore define **hyperparameters** as those parameters that characterize the Binary Classifier. Although they are also controllable (in a literary sense) we consider them separately to highlight the fact that they are not Prover-related. In practice by Hyperparameters we mean all those preprocessing techniques, enhancements and modifications that decades of Machine Learning have devised.

3.2. Goals

Our approach is defined and evaluated with two *goals* in mind. They are key drivers to the design of experiments aimed at validating the technique. The first goal **G1** is to achieve an adequate **detection capability**. In simpler terms, we want our models to successfully tell apart malicious activity from benign one. For a quantitative evaluation, we use the well-known evaluation metrics: Precision, Recall, F1 Score, Ac-

curacy. Our primary focus is on Recall and F1 score, although we also monitored the other two quantities for completeness. We are interested in Recall over Precision because raising a false alarm is a lot less dangerous than labelling a malicious counter as benign. Instead we prefer F1 Score over Accuracy because it provides a good estimate of the Classifier's general performance, and unlike Accuracy it is also reliable for imbalanced datasets. This consideration is helpful because our datasets are inherently imbalanced due to the technical limitations of Flash Memories. The second goal **G2** is to achieve G1 by introducing an acceptable computational overhead. The latter is dependant on the type of counters that you monitor and the mechanisms that you use for logging. The overhead is computed w.r.t. the *effective* attestation time, i.e. the time the AR would take to attest memory if it was not interrupted.

4. Experimental Validation

4.1. Experimental Setup

We show the setup for our experiments aimed at verifying whether G1 is attainable. *Section 4.2.4* illustrates how we varied this setup to test G2's attainability.

4.1.1 What to test? Why?

The answers are a natural consequence of *Section 3.2*. We monitor our target metrics on a subset of classifiers. All G1-related experiments investigate how the type of application running on the MCU (actually its metaparameters) affect the classification task. We analyzed two opposite extremes: using all counters vs dropping applicative counters. The latter is due to applicative counters being more demanding in terms of performance (see *Section 4.2.4*). This analysis was also complemented by a Feature Importance analysis, which we do not report here for sake of brevity. Finally, we are interested in understanding how the induced overhead is influenced by the available hardware and the type of counters that you choose.

4.1.2 Hardware/Software Setup

We evaluated our methodology on a STM32L552ZEQ development board. It incorporated a Cortex-M33 MCU with 512 KB of Flash Memory and 256 KB of SRAM. Despite its low price, it is endowed with ARM TrustZone [8] protection. We relied on the Data Watchpoint and Trace (DWT) Unit to count low-level events. This component offers 6 registers, each counting a specific microarchitectural event. We also monitored some high-level events, which we will present in a few lines. From a software standpoint, we implemented some tasks managed by the FreeRTOS microkernel. Tasks are in charge of mimicking the application's behavior and stimulate architectural counters. Each task stimulates one architectural counter, by executing instructions that trigger the corresponding events. However, they also indirectly stimulate other counters, though with minor impact. To each task we also associate an applicative counter. More precisely, we log each task iteration and store the total number in a dedicated counter. Finally we added two more tasks: one implementing the **Attestation Routine** and a **Malicious Task** in charge of emulating malicious activity.

4.1.3 Tested Classifiers

Testing every existing Classifier is obviously unfeasible. Thus we selected three models for our experiments: Logistic Regression (LR), Decision Tree (DT), and Support Vector Machine (SVM). **Logistic Regression** is a simple yet powerful *generalized linear model*. In spite of its linearity, using an appropriate feature expansion of the inputs allows to consider even non-linear relations. **Decision Trees** are used worldwide, and past works already showed their effectiveness in classifying Performance Counters for Malware Detection. **SVM** too showed its effectiveness in Malware Detection tasks, and it provides the remarkable benefit of considering complex features at little extra cost.

4.1.4 Handling Metaparameters

To validate our method in a broad range of settings, we tested several configurations of Entropy and Activity. The full space of these parameters is very large, so we sampled 16 configurations of the form $(a, e) \in \{\text{VERY\_LOW,LOW,MEDIUM,HIGH}\}^2$. Each of these labels represents concrete values of Ac-

tivity and Entropy. This mapping is achieved by modeling the applicative tasks' iterations as discrete random variables with a uniform distribution $U(x, y)$. We defined 16 tuples (x, y), one for each (a, e) configuration. We then tuned the (x, y) tuples to establish a partial ordering among Entropies and Activities. Once a configuration of Activity and Entropy levels was fixed, we ran several Attestation Routines. Some of them were interrupted by benign code only, others were subject to a malicious relocation. Because we want a Classifier that is able to detect a broad range of attacks, relocations were run by picking random Uncontrollable Parameters. While there are only two types of roving malware (transient and self-relocating), the space of malware sizes is quite large. We selected a reasonable subset based on researches in public malware repositories. After some preliminary results with high performance metrics, we reduced the tesed-sizes range to $[2KB, 16KB]$. This was meant to thoroughly stress our method and possibly highlight its limits.

4.1.5 Overhead Estimation Setup

The setup for Overhead Estimation was slightly different. We neglected both Controllable and Uncontrollable parameters, as the overhead is independent of them. We also activate the ARM TrustZone protection and ran the AR in the Secure World. This is meant to simulate a real-life deployment context in which hardware protection is required.

4.2. Experimental Results

This section presents our experimental results. We omit the Recall for the sake of brevity. However, since Recall showed a behavior that is quite similar to F1 Score, this is not a significant loss.

4.2.1 Full Set of Counters

Figure 2 shows the F1 Score in every Activity-Entropy configuration. Using the full set of counters shows promising results. DT has decent performance, despite being the worst-performing model. LR and SVM achieve high F1 and Recall scores, with SVM showing excellent results. The high scores obtained by SVM are likely caused by the high frequency of the applicative events that we log. This represents an extreme case, and we do not expect the average set of applicative events to provide detection rates just as high.

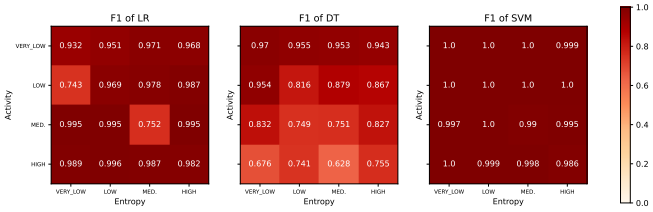


Figure 2: F1 score in every Metaparameter Configuration

4.2.2 Architectural Counters Only

After dropping applicative counters we obtain the scores in Figure 3. LR undergoes massive performance degradation. SVM is still a robust model, though it loses quite a bit in some challenging configurations. Finally DT is not significantly affected by dropping applicative counters. This setting represents yet another extreme case, but dual w.r.t. the previous one (that is, the case of applicative events occurring at frequency 0 Hz). We chose to show both extremes to provide a sort of upper and lower bounds to realistic detection power.

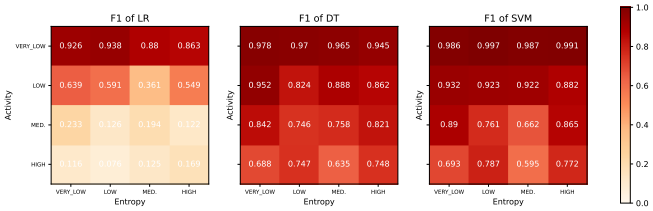


Figure 3: F1 score after dropping Applicative Counters

4.2.3 Enhancements

The performance degradation of LR and SVM when you drop architectural counters is quite unpleasant. Therefore we searched for possible enhancements to improve those situations. In fact, as reasoning on overhead shows, achieving satisfactory detection rates using only architectural counters is the ideal situation. Among all the tested techniques, spherical Box-Cox transformations [3] provide great improvement at little computational cost. Figure 4 shows the improvements obtained on LR. SVM also showed non-negligible improvements,though not

as sharp as LR. Labels of the form (λ, s) in each cell indicate that for that configuration the best test score s was the result of applying a transformation with hyperparameter λ.

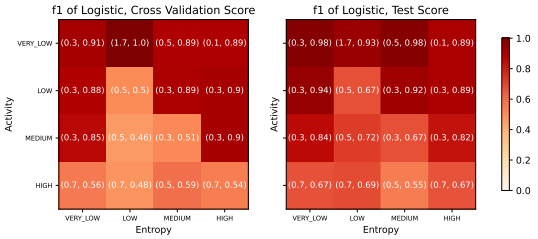


Figure 4: Validation (left) and Test (right) F1 score of LR using Box-Cox transformations

4.2.4 Overhead

Our primary focus is on Prover-side overhead. This is in line with the scientific literature on RA, which often assumes unbounded computational power for the Verifier. The best-case scenario is that in which you are able to achieve G1 with architectural counters only. We measured the corresponding overhead and plotted the measured results in Figure 5.

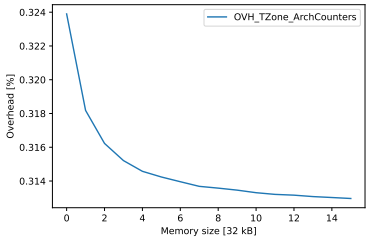


Figure 5: Experimental overhead using Architectural counters only. Sizes in the x-axis are represented as multiples of 32 KB

The overhead is very low and tends to zero as the attested memory size increases. This is a consequence of the fact that architectural counters are updated via hardware, and the only extra software operations are few writes and reads to activate, read and reset counters. Note that all these operations are performed in constant time. Some architectures might be subject to performance degradation due to **overflow**. In fact, some counters are implemented by registers with few bits, requiring to manually update the real counters' values on overflow. However in practice we have always witnessed that 32 bits are enough to avoid this issue. Finally, the most adversarial situation is that in which you need Applicative Counters to enhance the model's performance. A precise estimation of the overhead induced by applicative counters is a complex task that is beyond this work's scope. This is due to the several sets of events that you can choose to monitor. Intuitively, if one is able to achieve G1 by logging events that on average occur at low frequency, the overhead experiences a significant but acceptable increase. If instead you need high-frequency events, it is likely that the overhead becomes too cumbersome. By searching real-life use-cases [6, 7, 11], we found a decent number of examples characterized by low-frequency real life events. The latter prove that despite everything, using applicative counters can be a viable option at least for a non-negligible subset of practical use-cases.

5. Conclusions

We claim we achieved G1. The tested models showed high detection capabilities overall. Our experiments also prove that applicative counters are extremely beneficial, especially for Logistic Regression and SVM. Decision Trees are instead not particularly relying on them. Evidence shows that even if you lose some accuracy by dropping applicative counters, some enhancements can compensate for that. Specifically, Box-Cox transformations showed to be an easy and convenient method for that purpose. Despite causing possibly severe losses in detection power, using only architectural leads to low overhead. The latter increases when you introduce applicative counters. Logging low-frequency applicative events is likely to introduce moderate performance degradation. However, if you require high-frequency events to achieve G1, performance is likely to become unacceptable. This represents a possible limit of our approach. Overall, we noticed the emergence of a **detection-overhead** tradeoff. Logging high-frequency applicative events provides more detailed information for the Binary Classifier. However, at the same time this increases the computational overhead. Conversely, using low-frequency applicative events introduces smaller overhead, but might jeopardize the Classifier's performance. In real-life applications we recommend logging several events of interest at

various frequencies, and managing this tradeoff by selecting the event set that best suits your requirements.

6. Acknowledgements

To my family, for their constant struggle and sacrifices for my education and independence.
To those friends and beloved ones who have motivated me in the darkest times.
To Professor Zaccaria, for his guidance, and for showing me the beauty of Scientific Research.
To my other Mentors, for inspiring me to pursue my dreams.
Thank you.

References

[1] Mohammad Bagher Bahador, Mahdi Abadi, and Asghar Tajoddin. HPCMalHunter: Behavioral malware detection using hardware performance counters and singular value decomposition. In *2014 4th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 703–708, 2014.

[2] Alexander Sprogø Banks, Marek Kisiel, and Philip Korsholm. Remote Attestation: A Literature Review, 2021.

[3] Manuele Bicego and Sisto Baldo. Properties of the Box–Cox transformation for pattern classification. *Neurocomputing*, 218:390–400, 2016.

[4] Xavier Carpent, Norrathep Rattanavipanon, and Gene Tsudik. Remote attestation of IoT devices via SMARM: shuffled measurements against roving malware. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 9–16, 2018.

[5] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. VRASED: a verified hardware/software co-design for remote attestation. In *28th USENIX Security Symposium*, 07 2019.

[6] Roberto López-Blanco, Miguel A. Velasco, Antonio Méndez-Guerrero, Juan Pablo Romero, María Dolores del Castillo, J. Ignacio Serrano, Eduardo Rocon, and Julián Benito-León. Smartwatch for the analysis of rest tremor in patients with Parkinson's disease. *Journal of the Neurological Sciences*, 401:37–42, 2019.

[7] Pacific Marine Enviromental Laboratory OCS: Ocean Climate Station. Sampling rates in a weather monitor. `https://www.pmel.noaa.gov/ocs/sampling-rates`.

[8] Sandro Pinto and Nuno Santos. Demystifying ARM TrustZone: A comprehensive survey. *ACM Computing Surveys*, 51:1–36, 01 2019.

[9] Martin Rosso, Joost Renes, Nikita Veshchikov, Eduardo Alvarenga, and Jerry den Hartog. Actionable malware classification in embedded environments using hardware performance counters. In *SPACE 2021: Eleventh International Conference on Security, Privacy and Applied Cryptographic Engineering, SPACE 2021 ; Conference date: 10-12-2021 Through 13-12-2021*, December 2021.

[10] ST-Microelectronics. Artificial intelligence (AI) face recognition function pack for STM32Cube. `https://www.st.com/en/embedded-software/fp-ai-facerec.html`.

[11] Xueyi Wang, Joshua Ellul, and George Azzopardi. Elderly fall detection systems: A literature survey. *Frontiers in robotics and ai*, 7, June 2020.