POW-HOW: An enduring timing side-channel to evade online malware sandboxes

Antonio Nappa[‡][†][♦], Panagiotis Papadopoulos[○], Matteo Varvello[⋆], Daniel Aceituno Gomez[†], Juan Tapiador[†], Andrea Lanzi[⋄]

‡ UC Berkeley, † Universidad Carlos III de Madrid, ♦ Zimperium zLabs Team, ○ Telefonica Research, ⋆ Nokia Bell Labs, ⋄ University of Milan

Abstract. Online malware scanners are one of the best weapons in the arsenal of cybersecurity companies and researchers. A fundamental part of such systems is the sandbox that provides an instrumented and isolated environment (virtualized or emulated) for any user to upload and run unknown artifacts and identify potentially malicious behaviors. The provided API and the wealth of information in the reports produced by these services have also helped attackers test the efficacy of numerous techniques to make malware hard to detect.

The most common technique used by malware for evading the analysis system is to monitor the execution environment, detect the presence of any debugging artifacts, and hide its malicious behavior if needed. This is usually achieved by looking for signals suggesting that the execution environment does not belong to a the native machine, such as specific memory patterns or behavioral traits of certain CPU instructions.

In this paper, we show how an attacker can evade detection on such online services by incorporating a Proof-of-Work (PoW) algorithm into a malware sample. Specifically, we leverage the asymptotic behavior of the computational cost of PoW algorithms when they run on some classes of hardware platforms to effectively detect a non bare-metal environment of the malware sandbox analyzer. To prove the validity of this intuition, we design and implement the POW-HOW framework, a tool to automatically implement sandbox detection strategies and embed a test evasion program into an arbitrary malware sample. Our empirical evaluation shows that the proposed evasion technique is durable, hard to fingerprint, and reduces existing malware detection rate by a factor of 10. Moreover, we show how bare-metal environments cannot scale with actual malware submissions rates for consumer services.

1. Introduction

Malware attacks have a significant financial cost, estimated around $1.5 trillion dollars annually (or $2.9 million dollars per minute) [42], with predictions hinting at this cost to reach $6 trillion dollars by 2021 [30]. Due to the sheer amount of known malware samples [76,28], manual analysis neither scales nor allows to build any comprehensive threat intelligence around the detected cases (e.g., malware clustering by specific behavior, family or infection campaign). To address this problem, security researchers have introduced *sandboxes* [19]: isolated environments that automate the dynamic execution of malware and monitor its behavior under different scenarios. Sandboxes usually comprise a set of virtualized or emulated machines, instrumented to gather fundamental information of the malware execution, such as system calls, registry keys accessed or modified, new files created, and memory patterns.

As a next step, online services came to bring malware analysis from security experts to the common users [66]. Online malware scanners are not only useful for the users but also for the attackers. In fact by allowing an artefact to be checked multiple times against various state-of-the-art of malware analysis sandboxes, attackers can tune the evasiveness of their malware samples by exploiting the feedback reported by these services and try various techniques before making the sample capable of detecting the presence of a sandbox. Specific CPU instructions, registry keys, memory patterns, and *red pills* [70,67,58] are only a few of the signals used by attackers for identifying glitches of the emulated environment that can disclose the presence of a sandbox environment. These techniques have triggered an arms-race, with the more sophisticated web malware scanners rushing to spoof any such exploitable signals [43].

In this work, we show how an attacker can evade malware analysis of these scanning services by leveraging Proof-of-Work (PoW) [35] algorithms. Our intuition lies on the fact that, like NP-class problems [79], the asymptotic behavior of a PoW algorithm is constant in terms of computational power [35], e.g., CPU and memory consumption which remain stable over time. Accordingly, PoW algorithms are perfect candidates for benchmarking the computation capability of the underlying hardware. In such scenario the benchmark can be leveraged as a fingerprint of the underlying computing infrastructure, revealing the presence of a sandbox since it shows a statistical deviation compared with the native hardware platform. Moreover, current defensive techniques that aim at spoofing the virtualization signals present in contemporary sandboxes cannot act as countermeasures against the stable timing side-channels that our technique exploits.

A key advantage of using PoW techniques is that they are a time-proof and self-contained mechanism compared to other more fine-grained timing side-channel approaches that try to detect the underlying hardware machine. In fact, our system does not require access to precise timing resources for detecting the emulated environment (e.g., network or fine-grained timers). In our evaluation we empirically validate that a PoW-based technique can detect an emulated environment with high precision just by looking at the output of the algorithm (i.e., execution time, and number of successful iterations). Furthermore, PoW implementations do not raise any suspicion to automated malware sandboxes compared with the stalling code (e.g., infinite loops and/or sleep) that is easier to detect because of CPU idleness [52]. Fingerprinting PoW algorithms as a malware component is feasible e.g., by checking the usage of particular cryptographic instructions. However, using it as a proxy signal for detecting malware would produce a large number of false positives since PoW algorithms are part of legitimate applications such as Filecoin [68] and Hashcash [9].

Contributions. In this paper, we make the following contributions:

1. We design and implement POW-HOW: a framework to automatically create, inject, and evaluate PoW-based evasion strategies in arbitrary programs. POW-HOW operates as a three-step pipeline. First (step 1) multiple PoW algorithms are thoroughly tested across different hardware platforms (Raspberry Pi 3, Dual Intel Xeon, Intel i9), operating systems (Linux Ubuntu 18.03 and Windows 10), and machine loads. The outcome of these tests (step 2) is used to build a statistical characterization of each PoW's execution time under each setting. We use the Bienaymé–Chebyshev inequality [11] to obtain statistical evidence about the expected execution time. Next,

a miscreant can upload its malware to the POW-HOW framework and select the evasion mechanism to be used. Finally (step 3), POW-HOW automatically evaluates the accuracy of the evasion mechanism selected and embedded in the uploaded malware via several tests on multiple online sandbox services [66].

2. We empirically evaluate each step of POW-HOW's pipeline. For the PoW threshold estimation, we have tested three popular PoW algorithms (Catena [27], Argon2 [20,21] and Yescrypt [10]) using multiple configurations. During 24 hours of testing, we find Chebyshev inequality values higher than 97% regardless of PoW and setting. This result verifies high determinism in PoW execution times on real hardware, thus validating the main intuition behind this work. We test our technique on top of two known ransomware families by submitting to three sandboxes several variants that include PoW-based evasion. The results demonstrate how PoW-based evasion reduces the number of detections, even in the presence of anti-analysis techniques such as code virtualization or packing.
3. To further quantify the efficacy of PoW-based evasion with real-world sandboxes, we wrote a fully functional malware sample, integrated with an evasion mechanism based on Argon2, and submitted it to several online sandboxes. All the reports from each sandbox mark our malware as *clean*. We further discuss the behavioral analysis for our malware, as well as potential countermeasures to this novel PoW-based evasion mechanism we have proposed. To ensure the reproducibility of our results and foster further research on this topic, we make the source code of our system publicly available [16][1].

2. Background

2.1 Malware and Malware Analysis

Together with the evolution of malicious software, researchers and professionals have tried to improve their tools and skills to understand malware and counter its consequences. There is a huge amount of literature devoted to analyze and counter malware [62,80,39,47,24,77,46,60]. Every aspect of this phenomenon has been taken into consideration, from its network infrastructure, to the code that gets reused among samples, unexplored paths in the control-flow, sandbox design and instrumentation. Nonetheless the arms race keeps running, while new analysis evasion techniques are found, new countermeasures get developed.

Anti-Analysis Techniques: There are several anti-analysis techniques which have been developed during the years by miscreants, and promptly countered by our community: e.g., packers [56,75], emulators [71], anti-debugging and anti-disassembly tricks and stalling code. Among all these techniques the only one that seems to resist is stalling code, which is very difficult to detect [50]. Indeed, over 70% of all malware attacks involved evasive zero-day malware in Q2 of 2020: a 12% rise on the previous quarter [29]. This denotes that evasive malware is a phenomenon that will hardly disappear and there will always be continuous research in evading analysis systems.

[1] `https://github.com/anonnymousubmission/Esorics2021_Paper159`

2.2 PoW for Malware Analysis Evasion

Proof-of-Work (PoW) [35] is a consensus mechanism that imposes computation workload on a node. A key feature of such algorithms is their asymmetry: the work imposed on the node is moderately hard but it is easy for a server to check the computed result. There are two types of PoW protocols: (a) *challenge-response* protocols, which require an interactive link between the server and the client, and (b) *solution-verification* protocols, which allow the client to solve a self-imposed problem and send the solution to the server to verify the validity of the problem and its solution. Such PoW protocols (also known as CPU cost functions) leverage algorithms like hashcash with doubly iterated SHA256 [51], momentum birthday collision [49], cuckoo cycle [73], and more.

In POW-HOW we use Argon2, which guarantees that by using the same input parameters, the amount of computation performed is asymptotically constant; hence, the variance of Argons2's execution time T is very small on the same platform. Moreover, Argon2 is based on a memory-hard function which, even in the case of parallel or specialized execution (e.g., ASICs or FPGAs), will not enhance scalability, and hence remains computationally bounded due to its asymptotic behavior.

The Argon2 algorithm takes the following input:

- A message string P, which is a password for password hashing applications. Its length must be within 32-bit size.
- A nonce S, which is used as salt for password hashing applications. Its length must be within 32-bit size.
- A degree of parallelism p that determines how many independent (but synchronized) threads can be run. Its value should be within 24-bit size (minimum is 1).
- A tag, which length should be within 2 and 32-bit.
- A memory size m, which is a number expressed in Kibibytes.
- A number of internal iterations t, which is used to tune the running time independently of the memory size. Its value should be within 32-bit size (minimum is 1).

These input parameters are used in our framework to define the computational boundary of the algorithm execution on a specific class of hardware machines. Once the parameters are set, the output of the PoW algorithm only depends on the hardware platform.

2.3 Side-channel Measurement

Various techniques have been proposed to detect if applications are running inside a sandbox/virtualizer/emulator. The most reliable of them is based on timing measurements [45]. Indeed, fine grained timers help also to build micro-architectural attacks such as Spectre and Meltdown [54,44]. The intuition behind our work is that PoW algorithms offer strong cryptographic properties with a very stable complexity growth, which make the approach very resilient to any countermeasure, such as using more powerful bare-metal machines to enhance performance and reduce the space for time measurements.

By exploiting the asymptotic behavior of the PoW algorithms, we build a statistical model that can be used to guess the class of environment where the algorithm is running and consequently distinguish between physical and virtualized, emulated or simulated architectures, like different flavors of malware sandboxes. Indeed, even fine grained red-pills techniques [67] such as CPU instruction misbehavior can be easily fixed in the sandbox or spoofed to thwart evasion techniques. On the other hand PoW stands on top of well defined mathematical and well defined computational behavior. Moreover, a simple modification of the PoW library avoids the malware sample to be fingerprinted by static techniques. If we take as an example of PoW complexity the one that is run in the crypto currency environment, we know that by design the computation complexity of the algorithm is increased for each new block of the blockchain transaction [61]. Such an increase of computation shows the asymptotic behavior that can be exploited by our technique. By applying PoW as a malware sandbox evasion technique, we get an off-the-shelf technique which improves the malware resilience and limits its analysis.

3. Our Approach: PoW-How

This section describes our threat model before describing our approach in detail. We first provide an overview of the technique (Section 3.2) and its main workflow. We then describe how the key parameters are estimated (Sections 3.3 and 3.4) and how an arbitrary sample can be equipped with the evasion module (Section 3.5).

3.1 Threat Model

In this paper, we assume a malware scanning service based on virtualized or emulated sandboxes, which allows users to upload and scan their individual files for free as many times as they need. Such a service joins together results from various state-of-the-art malware analysis sandboxes before responding back to the user with a detailed report about the detection outcome of each and every sandbox scanner used.

On the other hand, we assume an attacker who developed a program that includes (i) some malicious payload along with (ii) a technique to pause or alter the execution of the malicious program itself, when a possible malware analysis environment is detected. Before distributing the malicious program to the victims, the attacker may use a malware scanning service to assess its evasiveness.

3.2 System Design

As described in Section 2., PoW puzzles have moderately high solving cost and a very small verification time, like problems in the NP complexity class [79]. This implies that their asymptotic behavior is constant in terms of computational cost [35], e.g., CPU and memory consumption. PoW-How exploits this asymptotic behavior to build a statistical model that can be used to identify the class of hardware machines where the algorithm is running. Such a model can later be used to distinguish between physical and virtualized architectures, like those present in malware sandboxes. PoW-How is a three-step pipeline (see Figure 1):

1. *Performance Profiling*. It executes multiple PoW algorithms on several hardware and operating systems using different configuration settings and system loads.
2. *Model estimation*. The previous step provides the system with a measurement of the amount of time needed to execute the PoW on real hardware. By using the Bienaymé–Chebyshev [11] inequality, it then estimates the time (threshold) expected for a particular configuration to run on a given architecture.
3. *Integration*. Once the models are built, a malware developer can select a specific PoW and parameters to associate with an arbitrary malware sample. PoW-How

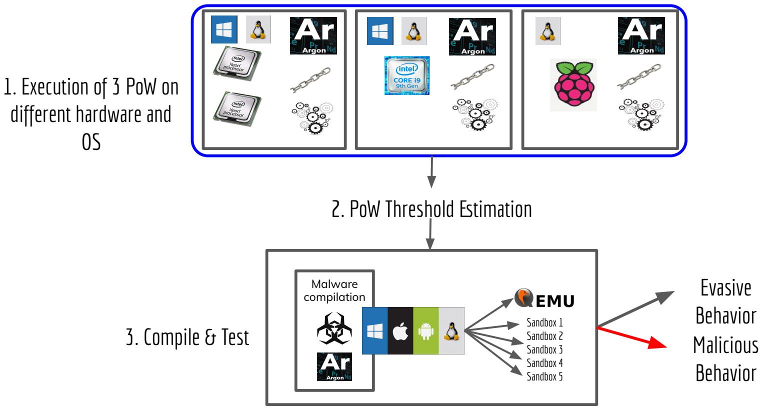

Fig. 1: High level overview of PoW-How. Step 1: execution of the PoW on several hardware/OSes using different configuration settings and system load. Step 2: threshold estimation based on execution time per configuration/architecture. Step 3: malware integration and test.

then generates a module with the chosen PoW, which is integrated with the sample by building a single statically-linked executable.

As ground truth, our methodology leverages a custom Cuckoo Sandbox [40] and popular crowd-sourced malware scanning services (like VirusTotal or similar [66]), as a testbed to report on the accuracy of the evasiveness of the malware in real-world settings.

3.3 Performance Profiling

The first step in PoW-How's pipeline produces a number of PoW executions using different algorithms, parameters, hardware, operating systems, and load settings:

Hardware: PoW-How leverages three machines representative of low, medium, and high-end platforms. The high-end machine is a desktop equipped with an Intel(R) Core(TM) i9-9900X CPU @ 3.50GHz with 10 physical cores and 20 threads equipped with a PCI-e M2 512GB disk and 32 GB of RAM. The medium-end machine is a workstation equipped with a Dual Intel(R) Xeon(R) CPU E5-2643 0 @ 3.30GHz with 16 physical cores and 64GB of RAM. Finally, the low-end device is a Raspberry Pi 3 which comes with a quad core ARMv7 Processor rev 4 (v7l) and 1GB of RAM.

Systems and loads: With the exception of the Raspberry Pi 3, the other hardware platforms are setup in dual boot, supporting both Linux Ubuntu 18.04.3 (64 bits) and Windows 10 (64 bits). Each platform can be further configured in *idle* and *busy* mode. The latter is achieved using `iperf` [34] a CPU bound network traffic generator to keep the operating system and the CPU occupied.

PoW and parameters: PoW-How currently supports three popular PoW algorithms: Catena [27], Argon2 [20,21], and Yescrypt [10]. Each PoW algorithm is executed multiple times with different input parameters on each hardware platform, operating system, and load setting. The parameters of each algorithm allow to control the amount of memory, parallelism, and complexity of the PoW. Our selection is based on common configuration of COTS hardware devices, with respect to memory and CPU. However, not all the selected algorithms have these parameters available for tuning and in some cases, their tuning is more coarse grained [27].

3.4 Threshold Estimation

The second step in PoW-How's pipeline aims at estimating the PoW thresholds for different settings (PoW algorithm, parameters, hardware, operating system, and load). This is achieved through a statistical characterization of the execution time in each setting using the Bienaymé–Chebyshev inequality [11]. This is a well-known result in probability theory stating that for a large class of distributions, no more than $\frac{1}{k^2}$ values of a distribution X can be more than k standard deviations (σ) away from the mean (μ):

$$Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2} \tag{1}$$

Using the empirical distribution of execution time observed in the previous step, this inequality allows us to select a threshold T (*i.e.,* a maximum execution time) which guarantees a high sample population coverage. The previous deduction enables us to determine with high probability the time T it will take for a PoW to run if the underlying platform is not virtualized. To reduce false positives, the evasion rule can be generalized to "the execution environment is virtualized if the PoW does not complete N executions in less than T seconds."

3.5 Malware Integration and Testing

The final step in PoW-How's pipeline is PoW integration with a malware sample provided as input. At this step, the attacker can upload its sample to PoW-How and select the PoW-based evasion mechanism to be used, along with its parameters. PoW-How further informs the attacker about the predicted accuracy of this selection.

PoW-How integrates the uploaded malware with the PoW selected and the Boost C++ libraries [65], which ease the OS interaction to build a single statically-linked executable. The compilation stage is automated as an Ansible [69] playbook and clang [55]. The integration is achieved at linking stage, so the malware will have a stub call to an external symbol that will be linked with the chosen PoW. PoW-How's pipeline then starts the Ansible scripts, which runs some tests and launch the compilation of the final binary for multiple platforms automatically.

Testing: To evaluate the accuracy of the newly generated evasion mechanism, we rely both on a local sandbox—a custom Cuckoo Sandbox [40] equipped with Windows 10 (64 bits), which is the most targeted OS for malware campaigns [78]—and several on-line free-of-charge sandbox services [66]. Once this step is completed, PoW-How offers to the user access to the set of reports generated by each sandbox.

4. Evaluation

In this section, we evaluate PoW-How's pipeline. We first analyze the combination of PoWs and their parameters currently supported by PoW-How. The outcome of this evaluation are the parameters N (cycle of execution made in less than T second) and T (maximum execution time) to be associated with the malware sample. We then discuss

Platform	Status	Win 10	Ubuntu 18.03
Intel i9	*idle*	4,500	9,325
	busy	3,642	8,867
Dual Intel Xeon	*idle*	6,005	7,897
	busy	4,320	7,012
Raspberry Pi 3	*idle*	-	300
	busy	-	143

Table 1: Number of consecutive PoW executions per hardware and OS combination over 24 hours. For a given platform, the first line refers to results obtained with the *idle* setting, while the second line refers to *busy* setting.

Garlic Graph Size	Min	Max	Sigma	Mean	K	Chebyshev
15	0.12	5.35	0.503	0.209	9.99	99.00%
18	1.13	35.61	4.22	1.86	7.94	98.41%
20	5.11	165.57	19.01	8.26	8.26	98.54%

Table 2: Statistical measurement results for Catena.

the accuracy of our evasion mechanism across using various case studies across three public malware scanning services: ([14], [12], [13]), along with our own Cuckoo Sandbox instance.

4.1 Threshold Estimation and PoW Algorithm Choice

For each PoW, we have selected different configurations with respect to memory footprint, parallelism, and algorithm internal iterations (see Tables 2 for Catena and 3 for Argon2i and Yescryot). Argon2i and Yescrypt have similar parameters (memory, number of threads, blocks) whereas Catena's only parameter is a graph size which grows in memory and will make its computation harder as the graph size increases.

PoW-How executes each PoW configuration on the low-end (Raspberry Pi 3), medium-end (Dual Intel Xeon), and high-end (Intel i9) machines. All PoW configurations are executed sequentially during 24 hours on each machine for both idle and busy conditions. As pointed out in Section 3., with the exception of the Raspberry Pi 3, all tests are performed on two operating system per hardware platform: Linux Ubuntu 18.04.3 (64 bits) and Windows 10 (64 bits).

Table 1 shows the total number of PoW executed over 24 hours per hardware, operating systems, and CPU load (*idle* or *busy*). Regardless of the CPU load on each machine, we observe two key insights. First, there is a significant drop in the number of PoW executions when considering Linux vs Windows, which is close to a 50% reduction in the high-end machine. This is due to operating system interaction, ABI and binary format, and ultimately idle cycle management. Second, a 30x reduction in the number of PoW executions when comparing high-end and low-end platforms, e.g., under no additional load the Raspberry Pi 3 completes 300 executions versus an average of 8,611 executions on both the high and medium-end machines. Finally, extra load on the medium and high-end machines causes a reduction in number of proofs computation of about 6-10%, averaging out to 7,300 executions between the two machines. A more dramatic 50% reduction was instead measured for the Raspberry Pi 3.

Next, we statistically investigate PoW execution times by mean of the Bienaymé–Chebyshev inequality (see Section 3.4). To balance equally sized datasets, we sampled 150 random executions (*i.e.,* the total number of executions that were possible to complete on the low-end platform) from the 9,325 executions available from both the medium and high-end platforms. Tables 2 and 3 show for each PoW and configuration, several statistics (min, max, σ, and K, Chebyshev inequality) of the PoW execution

Thr.	It.	Mem.	Min	Max	Sigma	Mean	K	Cheb.
1	10	1KB	0.01	0.70	0.09	0.02	7.9	98.4%
8	100	4KB	0.20	9.28	1.07	0.46	8.1	98.3%
16	500	8KB	2.03	88.8	10.5	3.85	7.9	98.4%
1	1K	8KB	0.00	0.02	0.00	0.01	6.1	97.3%
8	2K	32KB	0.03	0.56	0.05	0.05	10.5	99.1%
16	4K	64KB	0.08	5.00	0.51	0.19	9.4	98.9%

Table 3: Statistical measurement results for Argon2i (top) and Yescrypt (bottom). Thr. = number of threads. It. = number of algorithm steps. Mem. = amount of memory used in KiB. Cheb. = Chebyshev coverage.

Test	Relec	Forbidden Tear	Hello World
Original	23/72	26/72	3/72
Original+Code Virtualizer	32/72	n/a	19/72
Original+Themida	33/72	21/72	17/72
Original+PoW+Code Virtualizer	29/72	n/a	0/72
Original+PoW+Themida	32/72	18/72	9/72
Original+PoW	**3/71**	**3/72**	**2/72**

Table 4: Online Sandbox detection results for 2 ransomware samples (Relec and Forbidden Tear) and a benign test program using various anti-analysis configurations.

time computed across hardware platforms, OSes (when available), and load condition (idle, busy). Overall, we measured Chebyshev inequality values higher than 97% regardless of the PoW and its configuration. This confirms high determinism in the PoW execution times on real hardware, validating the main intuition behind this work.

Algorithm choice: The results above provide the basis to select a PoW algorithm along with its parameters to integrate with the input malware sample. These results indicate that PoW selection has minimal impact on the expected accuracy of the proposed evasion mechanism. We then selected Argon2i (with 8 threads, 100 internal functions and 4KiB of memory) motivated by its robustness and maturity. We leverage the results from Table 3 (top, second line) to set the parameters N (PoW execution) and T (evasion threshold) of an Argon-based evasion mechanism. The table shows that $K = 8.1$ seconds allows a good coverage for the execution time population (98.3%). We opted for a more conservative value of $T = 10$ and further performed multiple tests on our internal Cuckoo Sandbox. Given that our Cuckoo Sandbox could not even execute 1 PoW with $T = 10$, we simply set $N > 1$. We will use this configuration for the experimentation described in the remaining of this paper.

4.2 Case Study: Known Malware

We first analyze the effect of adding our PoW-based evasion strategy to the code of two well-known ransomware samples: Relec and Forbidden Tear. The use of real-world malwares, which are well know and thus easy to detect, allows us to comment on the impact that PoW-based evasion has on *malware reuse*, the practice of recycling old malware for new attacks. We use POW-HOW to generate various combinations of each original ransomware with/without PoW-based evasion strategy, code virtualization[2], and packing offered by Themida, a well-known commercial packer [64]. We verify that all the malicious operations of the original malwares were preserved across the generated versions.

We submitted all malware variants to three online sandboxes for analysis and checked how many AV engines (antivirus products) flag each variant as malicious (see Table 4). In the case of Relec, adding code virtualization or packing, results in more AV engines detecting the sample as malicious. This is likely due to the engines flagging such protections, not the malware sample itself. In all cases, the addition of PoW

[2] This cannot be applied to ForbiddenTear since it is written in .NET.

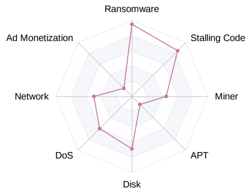

Fig. 2: Behavioral map of the malware PoC *without* PoW and *without* full static protection enabled.

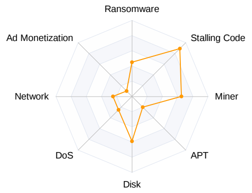

Fig. 3: Behavioral map of the malware PoC *without* PoW and *with* full static protection enabled.

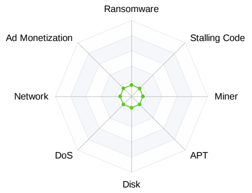

Fig. 4: Behavioral map of the malware PoC *with* PoW and *with* full static protection enabled.

decreases the number of detections by a factor of 10 [17], reaching a level where the difference between the label *malicious* and *false positive* is evanescent.

Table 4 also show results when submitting several variants of a standard Hello World program. Note that the original code has been flagged as malicious by 3 AV engines, though as it is possible to see from the report the detections are mislabeled *i.e.,* Relec is not recognized. This false positive could be due to a large number of submissions of the same code hash (due to its simplicity and popularity), our source IP being flagged, and other unknown factors which may influence the scoring. The table also shows that adding code virtualization or packing translates into a substantial increase in false positive detections even of a simple Hello World program, confirming our intuition above. Instead, adding our PoW-based evasion strategy results in less false positives, one less than the original code. This is likely due to the fact that our code on top of Hello World has more entropy, respect to a very simple one line program, looking more legit to engines that measure such kind of parameters.

Overall, these three case studies show that a PoW-based evasion strategy reduces the number of detections by 10x with known malware by preventing the sample from executing in the analysis sandbox. This result demonstrates large potential for malware reuse by coupling it with PoW-based evasion strategy. In the next section, we perform more controlled experiments based on *fresh* (*i.e.,* previously unseen) malware.

4.3 Case Study: Fresh Malware Sample

In order to further explore the results obtained in the previous case studies, we wrote a simple malware PoC (roughly 150 LoC) for Windows 10 (VC++) and Linux (C++). Our malware sample implements a basic ransomware functionality which scans the entire hard drive and encrypts all its files. This behavior should be easy to detect by any malware scanning services.[3] Using PoW-How, we automatically embed a PoW (Argon2i, as we will discuss below) and make sure to exhibit its malicious activity only if the PoW is successfully executed at least N times before a timeout T. Finally, we submitted different variants of our malware sample (with PoW, without PoW, with static sanitization) to several on-line sandboxes and the results were disheartening (see Table 5). For the static sanitation we remove the symbol tables and debugging symbols. Note that very similar results were also achieved with our local Cuckoo Sandbox. It

[3] The malware detection report for this malware without our PoW-based evasive measure has been anonymized [1,15].

is important to note that to check the execution of the malware payload we insert a *create-file* function at the beginning of the malware payload itself. Such file creation is visible on the behavioral report of the analyzed sandboxes in case the malware payload is executed[4]. We used such a simple test to check whether the PoW algorithm detects the emulated environment and so validate our technique. In case such a file is not present on the behavioral report, it means the PoW algorithm detects the emulated environment and stops the payload execution. None of the analyzed sandboxes is able to execute more than 1 PoW during $T = 10$ (or even $T = 20$ sec), which is worse than what a Raspberry Pi 3 can do even in presence of some extra load (e.g., see max value in the top of Table 1).

We made all the reports of our analysis publicly available, including screenshots of evasive malware samples [5]. It has to be noted that not all sandboxes report are the same, but they all signal the hard drive scan (Ransomware behavior) without full static protection (i.e., with the default compiler options). In Table 5 the number of PoW executed is visible only if a screenshot of the sandbox is available. As for the sandbox execution timeout, not all the analysis services had it available for selection.

Detection Rate Decrease: As it is possible to see POW-HOW's approach is capable of reducing to zero the detection rate of roughly 70 antiviruses run by the tested sandboxes [13,14,12] for any sample that we have tested. We have investigated the multiple facets of our technique (static and dynamic). Thus we conclude after looking also at the behavioural results of our samples that the whole technique is capable of reducing the detection rate to zero. The behavioural part plays a fundamental role as it is possible to see from the Hello World example and the behavioural maps generated by AV labels of Figures 2-4.

5. Security Analysis

The results shown in the previous section demonstrate that a POW-HOW-ed malware can effectively detect a sandbox and abort the execution of any malicious payload. This strategy is effective in getting a malware sample marked as "clean" by all sandboxes tested by POW-HOW (see Table 5). **POW-HOW's technique is simple to deploy, it does not require precise timing measurements and, thanks to its algorithmic properties, it will last for many years as a potential threat.**

We next discuss in detail the *behavioral* analysis of our malware. This is an analysis produced by a sandbox related to how a malware interacts with file system, network, and memory. If any of the monitored operations matches a known pattern, the sandbox can raise an alarm.

Figures 2, 3, and 4 show the behavioral analysis of our malware on a radar plot, labelled with most prevalent AV labels. The samples were submitted with different combinations of PoW and static protection. In Figure 2, the radar plot is mostly "green" (benign) with respect to some operations like phishing, banker and adware for which we would not expect otherwise. However, four "suspicious" (orange) behaviors are re-

[4] This reference has been anonymized not to violate the terms of service of sandbox vendors [1]

[5] The references have been anonymized not to violate the terms of service of sandbox vendors [2,5,6,7,3,8,4]

Sandbox	Evasion Timeout	PoW Timeout	# of PoW executed	Timeout	Notes
Sandbox1	10 secs	50	1	120	Clean
Sandbox1	15 secs	45	1	180	Clean
Sandbox1	20 secs	40	1	240	Clean
Sandbox1	20 secs	15	1	500	Clean
Sandbox2	20 secs	15	0	N/A	Clean
Sandbox3	20 secs	45	N/A	N/A	Clean
Sandbox3	20 secs	15	N/A	N/A	Clean

Table 5: Execution results of a custom ransomware sample on various sandboxes

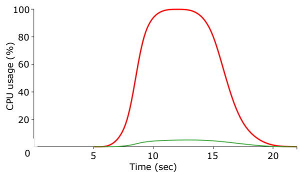

Fig. 5: CPU consumption of our malware PoC (Argon2d) Malware:red line, System Idle (PID 0):green line.

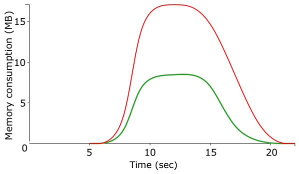

Fig. 6: Memory consumption of our malware PoC (Argon2d) Malware:red line, System Idle (PID 0):green line.

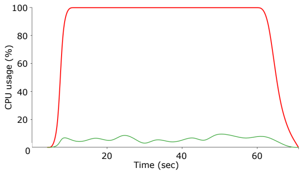

Fig. 7: CPU consumption of our malware PoC. T=60 seconds and 0.5 seconds between each PoW execution. Malware:red line, System Idle (PID 0):green line.

ported with respect to evader, spyware, ransomware, Trojan operations. While our malware PoC is not labeled as "malicious" (red), the suspicious flags for our binary would trigger further manual analysis that coukd reveal its maliciousness. It is thus paramount to investigate and mitigate such suspicious flags.

Our intuition is that the suspicious flags are due to the fact that our malware is neither packed nor stripped, and hence some of its functionality *i.e.,* exported functions, linked libraries, and function names are visible through basic static analysis that is usually also implemented in the dynamic sandbox environment. Accordingly, we strip out the whole static information from our binary and resubmit it as a new binary. Figure 3 shows the behavioral analysis of our PoC malware without PoW-based sandbox detection but with full static protection enabled. As expected, various signals have dropped from the behavioral report. Finally, Figure 4 shows the result of adding PoW to the last binary. A completely green radar plot which does not raise any suspicion illustrates the evasion effect of PoW-How.

CPU and memory usage: The main downside of associating a PoW with a malware sample is an increase in both CPU and memory consumption. We here report on CPU and memory consumption as measured by our sandbox. Figures 5 and 6 compare, respectively, CPU and memory utilization of our malware (red line) with System Idle (PID 0). With respect to CPU usage, the PoW associated with our malware causes an (expected) 100% utilization for the whole duration of the PoW ($T = 10$ sec). With respect to memory utilization, our malware only requires about 17 MB versus the 7 MB that utilizes a sample system process like System Idle (PID 0). This is a minor increase, unlikely to raise any suspicion.

Next, we investigate whether we can reduce the CPU usage of our PoC ransomware by setting a longer T (e.g., 60 sec) and a sleep of 0.5 sec between each PoW execution. Despite such sleeps, Figure 7 still shows 100% CPU utilization for the whole T (60 sec in this test). The lack of CPU reduction associated with the extra sleeps is counter-intuitive. The likely explanation is that the sandbox leverages a coarse CPU monitoring tool and, thus, the CPU reduction associated with our extra sleeps gets averaged out. These results provide a foundation to detect evasion techniques based on PoW. A sandbox could attempt heuristics based on a binary's CPU and memory consumption. We argue, however, that this is quite challenging because of the potential high number of false positives that can be generated.

6. Countermeasures

Evasion techniques are easily comparable with other anti-analysis techniques like *packing*. Packing techniques have evolved to such sophistication that it has become practically impossible to unpack a malware sample without dynamically executing it [41,75]. However, dynamically executing a sample can indeed trigger evasion techniques like stalling code. To counter evasion techniques, and especially the ones that POW-HOW implements, one idea would be to fingerprint the algorithms, e.g., CPU and memory footprint. However, it would be very easy for attackers to apply code polimorphism techniques and produce variants that diverge from the original implementation, as it is done with packers. This will constitute a challenge for the sandbox, which could generate a false negative by not being able to spot the algorithm. In Table 4, the Hello World program is detected as malicious and our technique reduces its detection rate and with a code virtualizer it makes the sample completely stealth.

Fingerprinting evasion: A common solution against red pills [67] is to reduce the amount of instructions failing due to emulation. As Martignoni et al. [57,58] show, the analysis can be automated and the fixes can be easily produced. However, with PoW the computational model is not seeking for emulation/virtualization failures or malfunctions. Instead, PoW is acting as a probe to spot a side channel in the execution time of the algorithm, which in this case is time-based.

Virtualized instructions set: Native execution of the cryptographic instructions is another potential countermeasure that could be considered to mitigate our approach. In such a case, the cryptographic instructions of the PoW algorithm are not emulated by the sandbox environment, but directly executed on the native CPU. Avoiding the emulation of the cryptographic instructions could clearly improve the computational performance of the PoW algorithm and reduce the success probability of the evasive behavior showed by POW-HOW. The technique described in the Inspector Gadget paper [46], which works at the program analysis level, may also work to avoid the execution of our evasion code. Once the sample is unpacked, it would be possible to extract and execute only the malware branch of the code as a gadget and analyze its behavior in isolation. However, a sufficiently complex packer or emulator would make such process very tedious and require manual effort, which makes this solution excessively complex to be implemented in an automated malware analysis service.

Specialized hardware: Even if our choice, Argon2, is resilient to specialized circuits for mining (ASICs and FPGAs), other PoW algorithms are not, and hence an analyst could equip his sandbox with a miner [74]. Such a dedicated hardware is expensive for a non-professional user (around $\$3,000$ at the time of writing). Nonetheless, if the phenomenon of sandbox evasion due to PoW proliferate, having such a platform would be of great help to offload the PoW calculations, through a tailored interface, and continue the execution of the malware sample inside the sandbox. The cost/benefit trade-off of adopting such a measure really depends on the intended scale of the analysis platform. For example, according to VirusTotal statistics [28], the service receives weekly more than 3M PE binaries. Hence, a dedicated hardware to defeat PoW evasion based techniques seem a good compromise, since it allows to analyze and discover new malicious behaviors.

Spoofing timers: The sandbox that gets a POW-HOW-ed malware could try to delay the time, which could mean to make our $T = 10$ seconds last much longer to achieve the payload execution. This approach may work well. Though, if we expect a total of at least 50 PoW iterations (see Section 3.4) and the sandbox is not able to execute more than one in about a minute for a unique malware sample, the analysis would take more than one hour. This will eventually extract the payload that will then require extra work to be reverse engineered, understood, and fingerprinted. Hence, this approach may not scale in terms of time/cost for the large number of samples that online sandboxes analyze daily.

Bare-Metal Sandboxes: Using bare metal hardware represents a reasonable solution that might be adopted within corporate companies but it is not possible to use such technology at Internet scale, *i.e.,* cloud-based solutions like Virus Total. Also, isolated sandboxes do not benefit of the information that on-line in cloud services have which leverages large scale cross-correlations.

7. Discussion

7.1 Ethical Considerations

The results obtained by POW-HOW regarding the analyzed publicly available sandboxes, normally used by malware analysts under their term of service (ToS), demonstrate that our technique works consistently either in our custom Cuckoo Sandbox implementation or in proprietary solutions. Our aim, though, is not to disrupt any business nor to difficult the operation of companies that profit from providing malware behavior analysis. We contacted all the platforms and vendors that we have tested with POW-HOW and we notified them about our findings. Part of the vendors were very positive and agreed to further collaborate to work on practical countermeasures. Unfortunately, the response we received from other vendors opposed any dissemination of our results, adopting a shortsighted security-through-obscurity approach which is not novel in our community. Consequently, tested vendors have been anonymized to avoid violation of their ToS. We purposely maintained the number of new variants submitted to the bare minimum, but our approach may transform easily any existing sample into a new one. The authors are available for contact for further information disclosure.

7.2 Bare-Metal Environments

In [43] the authors present BareCloud a bare-metal system which helps to detect evasive malware. This system in order to execute malware trades visibility against transparency. In other words it makes the analysis system transparent (non-detectable by malware) and produces less powerful analysis data (limited instrumentation). Indeed their detection technique leverages hierarchical similarity [36] comparison between different malware execution traces (virtualized and emulated) systems i.e., (Ether [32], Anubis [19], and VirtualBox [40]). One of the biggest problem of hierarchical similarity algorithms is scalability, which means that the algorithm should be polynomial in time and space. An example [63] of application and analysis of hierarchical similarity for binary program comparison shows $O(n^2)$ complexity. Hence using BareCloud as a production system for example for VirusTotal which claims [28] about 1.5M daily submissions means that the hierarchical comparison would approximate 2.250 billion of operations daily to detect evasive malware with bare metal equipment. It is evident that BareCloud can be useful in special cases, as briefly stated above, where also a manual analyst can make the difference. For the sake of scalability though virtualization and emulation methods cannot be fully replaced, even if it would be possible to instrument in hardware an entire system [52], the approach would suffer many other issues, for instance having a lot of physical hardware and maintaining it.

7.3 Economical denial of sustainability

Online sandboxes, like any other business, have costs to sustain. Ignoring evasive malware to avoid an additional cost is (for now) understandable. Unfortunately, malware that exploits POW-HOW's technique implies additional energy and memory costs, especially if submitted in large scale to such systems, opening avenues to EDoS attacks, which will try to make the on-line service not sustainable economically. These on-line services receive on average 1.5M samples daily. It is not difficult to imagine how much energy just a tenth of the total submissions can consume if it is running PoW. Such algorithm is one of the most energy intensive operation that a computer can perform. For instance, the yearly energy consumption of Bitcoin's blockchain is comparable to the one of a country such as Tunisia or Czech Republic [31]. We strongly recommend that not all evasion techniques are the same, and every technique that exploits hardware consumption side channels should be properly analyzed to avoid service disruption.

8. Related Work

There is a significant body of research [81,23,72,25,48,38,71] focusing on both designing novel evasion techniques for malware and also providing mechanisms to detect them. We next discuss the most relevant works related to ours.

Fingerprinting emulated environments: By recognizing the sandboxes of different vendors, malware can identify the distinguishing characteristics of a given emulated environment and alter its behavior accordingly. The work in [70] introduced the notion of *red pill* and released a short exploit code snippet that could be used to detect whether the code is executed under a VM or in a real platform. In [67], the authors propose an automatic and systematic technique (based on EmuFuzzer [57]) to generate red pills for detecting whether a program is executed inside a CPU emulator. In [58], the authors build KEmuFuzzer, which leverages protocol-specific fuzzing and differential analysis. KEmuFuzzer forces the hosting virtual machine and the underlying physical machine to execute specially crafted snippets of user- and system-mode code before comparing their behaviors. In [22] authors presented AVLeak, a tool that can fingerprint emulators running inside commercial antivirus (AV) software, which are used whenever AVs detect an unknown executable. The authors developed an approach that allows them to deal with these emulators as black boxes and then use side channels for extracting fingerprints from each AV engine. Instead, we show that even with completely transparent analysis programs, the real environment can be used by the malware to determine that it is under analysis. In [59] authors propose a ML-based approach to detect emulated environments. This technique is based on the use of features such as the number of running processes, shared DLLs, size of temporary files, browser cookies, etc. These features are named by the authors "wear-and-tear artifacts" and are present in real system as opposed to sandboxes. The authors use such features to train an SVM classifier. We also rely on modeling a distinguishing feature, in our case is a time channel arising from the asymptotic behavior of a Pow, not the presence or absence of system artefacts.

In [37], authors introduce the virtual machine monitor (VMM) detection and they propose a fuzzy benchmark approach that works by making timing measurements of the execution time of particular code sequences executed on the remote system. The fuzziness comes from heuristics which they employ to learn characteristics of the remote system's hardware and its configuration. In [26], the authors present a technique that leverages TCP timestamps to detect anomalous clock skews in VMs. A downside of the approach is that it requires the transmission of streams of hundreds of SYN packets to the VM, something that can be detected in the case of a honeypot VM and flagged as malicious behavior. Compared to the previous approaches, POW-HOW is more principled and offers a solid basis founded on cryptographic primitives (PoW) with a predictable and reproducible computational behavior on different tested platforms.

Detecting evasive malware: In [33], the authors propose Ether, a malware analyzer that eliminates in-guest software components vulnerable to detection. Ether leverages hardware virtualization extensions such as Intel VT, thus residing outside of the target OS environment. In [43], the authors present an automated evasive malware detection system based on bare-metal dynamic malware analysis. Their approach is designed to be transparent and thus robust against sophisticated evasion techniques. The evaluation results showed that it could automatically detect 5,835 evasive malware out of 110,005 tested samples. In [18], authors propose a technique to detect malware that deploys evasion mechanisms. Their approach works by comparing the system call trace recorded when running a malware program on a reference system with the behavior observed in the analysis environment. In [53], authors propose a system for detecting environment-sensitive malware by comparing its behavior in multiple analysis sandboxes in an automated way. Compared to previous techniques, our approach is agnostic to system artifacts and cannot be recognized by only monitoring the system operations.

9. Conclusion

Online malware scanning services are becoming more and more popular, allowing users to upload and scan artefacts against AV engines and malware analysis sandboxes. Common mechanisms used by malware samples to avoid detection include the inspection of signals that imply the existence of a virtualized or emulated environment. These strategies triggered an arms-race where online malware scanners patch such signals to make virtualization transparent. In this paper, we leverage PoW techniques as the basis for a novel malware evasion technique due to their ability to fingerprint real hardware. We provide empirical evidence of how it can be used to evade online malware analysis sandboxes and discuss potential countermeasures. The implementation of our approach goes beyond a simple proof-of-concept, showing that injecting evasion modules can be easily automated on any arbitrary sample. We make our code and results publicly available in an attempt to increase reproducibility and stimulate further research in this area.

References

1. : Evasive malware analysis report. anonymized (2020)
2. : Evasive malware analysis report. anonymized (2020)
3. : Evasive malware analysis report. anonymized (2020)
4. : Evasive malware analysis report. anonymized (2020)
5. : Evasive malware analysis report - 1. anonymized (2020)
6. : Evasive malware analysis report - 2. anonymized (2020)
7. : Evasive malware analysis report - 3. anonymized (2020)
8. : Evasive malware analysis sandbox. anonymized (2020)
9. Adam Back: Hashcash: antin-spam tool. http://www.hashcash.org/ (2020)
10. Alexander Peslyak, T.H.: yescrypt - scalable kdf and password hashing scheme. www.openwall.com/yescrypt (2015)
11. Alsmeyer, G.: Chebyshev's inequality. In: International Encyclopedia of Statistical Science. Springer Berlin Heidelberg (2011)
12. anonymized: Sandbox 1. anonymized (2020)
13. anonymized: Sandbox 2. anonymized (2020)
14. anonymized: Sandbox 3. anonymized (2020)
15. Antonio Nappa, Panagiotis Papadopoulos, Matteo Varvello, Daniel Aceituno Gomez, Juan Tapiador, Andrea Lanzi: PoC Behaviour (No Evasion) - anonymized. anonymized (2020)
16. Antonio Nappa, Panagiotis Papadopoulos, Matteo Varvello, Daniel Aceituno Gomez, Juan Tapiador, Andrea Lanzi: Artifact repository. https://github.com/anonnymousubmission/Esorics2021_Paper159 (2021)
17. Antonio Nappa, Panagiotis Papadopoulos, Matteo Varvello, Daniel Aceituno Gomez, Juan Tapiador, Andrea Lanzi: Relec + PoW + static sanitization) - anonymized. anonymized (2021)
18. Balzarotti, D., Cova, M., Karlberger, C., Vigna, G.: Efficient detection of split personalities in malware. In: Proc. 17th Annual Network and Distributed System Security Symposium (NDSS), 2010 (2010)
19. Bayer, U., Comparetti, P.M., Hlauschek, C., Krügel, C., Kirda, E.: Scalable, behavior-based malware clustering. In: NDSS. The Internet Society (2009), http://dblp.uni-trier.de/db/conf/ndss/ndss2009.html#BayerCHKK09
20. Biryukov, A., Dinu, D., Khovratovich, D.: Argon2: New generation of memory-hard functions for password hashing and other applications. In: IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016 (2016)
21. Biryukov, A., Dinu, D., Khovratovich, D., Josefsson, S.: Argon2 rfc. www.tools.ietf.org/id/draft-irtf-cfrg-argon2-05.html (2019)
22. Blackthorne, J., Bulazel, A., Fasano, A., Biernat, P., Yener, B.: Avleak: Fingerprinting antivirus emulators through black-box testing. In: 10th USENIX Workshop on Offensive Technologies (WOOT 16). USENIX Association, Austin, TX (Aug 2016), https://www.usenix.org/conference/woot16/workshop-program/presentation/blackthorne
23. Brengel, M., Backes, M., Rossow, C.: Detecting hardware-assisted virtualization. In: Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721. p. 207–227. DIMVA 2016, Springer-Verlag, Berlin, Heidelberg (2016)
24. Caballero, J., Grier, C., Kreibich, C., Paxson, V.: Measuring Pay-per-Install: The Commoditization of Malware Distribution. In: Proceedings of the 20th USENIX Security Symposium (2011)
25. Canali, D., Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: A quantitative study of accuracy in system call-based malware detection. In: Heimdahl, M.P.E., Su, Z. (eds.) International Symposium on Software Testing and

Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012. pp. 122–132. ACM (2012). https://doi.org/10.1145/2338965.2336768, https://doi.org/10.1145/2338965.2336768

26. Chen, X., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J.: Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: 2008 IEEE international conference on dependable systems and networks with FTCS and DCC (DSN). pp. 177–186. IEEE (2008)
27. Christian Forler, Stefan Lucks, J.W.: The catena password-scrambling framework. www.uni-weimar.de/fileadmin/user/fak/medien/professuren/Mediensicherheit/Research/Publications/catena-v3.1.pdf (2015)
28. Chronicle Security: File statistics during last 7 days. https://www.virustotal.com/en/statistics/ (2020)
29. Coker, J.: Evasive malware threats on the rise despite decline in overall attacks. https://www.infosecurity-magazine.com/news/evasive-malware-rise-decline/ (2020)
30. Cybersecurity Ventures: Global cybercrime damages predicted to reach $6 trillion annually by 2021. https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/ (2018)
31. Digiconomist: Yara Signature Detector. https://digiconomist.net/bitcoin-energy-consumption (2007)
32. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: Malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM Conference on Computer and Communications Security. p. 51–62. CCS '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1455770.1455779, https://doi.org/10.1145/1455770.1455779
33. Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: Proceedings of the 15th ACM conference on Computer and communications security. pp. 51–62 (2008)
34. Dugan, J., Elliott, S., Mah, B.A., Poskanzer, J., Prabhu, K.: iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/ (2020)
35. Dwork, C., Naor, M.: Pricing via processing or combatting junk mail. In: Proceedings of the 12th Annual International Cryptology Conference on Advances in Cryptology. CRYPTO '92, Springer-Verlag (1992)
36. Feldman, R., Dagan, I.: Knowledge discovery in textual databases (kdt). In: Proceedings of the First International Conference on Knowledge Discovery and Data Mining. p. 112–117. KDD'95, AAAI Press (1995)
37. Franklin, J., Luk, M., McCune, J.M., Seshadri, A., Perrig, A., Van Doorn, L.: Remote detection of virtual machine monitors with fuzzy benchmarking. ACM SIGOPS Operating Systems Review **42**(3), 83–92 (2008)
38. Graziano, M., Canali, D., Bilge, L., Lanzi, A., Balzarotti, D.: Needles in a Haystack: Mining Information from Public Dynamic Analysis Sandboxes for Malware Intelligence. In: Proceedings of the 24rd USENIX Security Symposium (USENIX Security) (August 2015)
39. Gu, G., Yegneswaran, V., Porras, P., Stoll, J., Lee, W.: Active botnet probing to identify obscure command and control channels. In: Proceedings of 2009 Annual Computer Security Applications Conference (ACSAC'09) (December 2009)
40. Guarnieri, C.: Cuckoo sandbox. https://cuckoosandbox.org/ (2010)
41. Haq, I.U., Chica, S., Caballero, J., Jha, S.: Malware Lineage in the Wild. Computers & Security **78**(C), 347–363 (August 2018). https://doi.org/10.1016/j.cose.2018.07.012
42. Infosecurity Magazine: Cybercrime costs global economy $2.9m per minute. https://www.infosecurity-magazine.com/news/cybercrime-costs-global-economy/ (2019)
43. Kirat, D., Vigna, G., Kruegel, C.: Barecloud: Bare-metal analysis-based evasive malware detection. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 287–301. USENIX Association, San Diego, CA (Aug 2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kirat

44. Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: 40th IEEE Symposium on Security and Privacy (S&P'19) (2019)
45. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology. p. 104–113. CRYPTO '96, Springer-Verlag, Berlin, Heidelberg (1996)
46. Kolbitsch, C., Holz, T., Kruegel, C., Kirda, E.: Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In: 31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA. pp. 29–44. IEEE Computer Society (2010). https://doi.org/10.1109/SP.2010.10, `https://doi.org/10.1109/SP.2010.10`
47. Kotzias, P., Bilge, L., Caballero, J.: Measuring PUP Prevalence and PUP Distribution through Pay-Per-Install Services. In: Proceedings of the 25th USENIX Security Symposium (2016)
48. Lanzi, A., Balzarotti, D., Kruegel, C., Christodorescu, M., Kirda, E.: Accessminer: using system-centric models for malware protection. In: Al-Shaer, E., Keromytis, A.D., Shmatikov, V. (eds.) Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010. pp. 399–412. ACM (2010). https://doi.org/10.1145/1866307.1866353, `https://doi.org/10.1145/1866307.1866353`
49. Larimer, D.: Momentum–a memory-hard proof-of-work via finding birthday collisions. Tech. rep. (2014)
50. Lastline Inc.: Not so fast my friend – using inverted timing attacks to bypass dynamic analysis. www.lastline.com/labsblog/not-so-fast-my-friend-using-inverted-timing-attacks-to-bypass-dynamic-analysis/ (2014)
51. Laurie, B., Clayton, R.: Proof-of-work proves not to work; version 0.2. In: Workshop on Economics and Information, Security (2004)
52. Li, L.W., Duc, G., Pacalet, R.: Hardware-assisted memory tracing on new socs embedding fpga fabrics. In: Proceedings of the 31st Annual Computer Security Applications Conference. p. 461–470. ACSAC 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2818000.2818030, `https://doi.org/10.1145/2818000.2818030`
53. Lindorfer, M., Kolbitsch, C., Comparetti, P.M.: Detecting environment-sensitive malware. In: International Workshop on Recent Advances in Intrusion Detection. pp. 338–357. Springer (2011)
54. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18) (2018)
55. LLVM: Clang: a c language family frontend for llvm. `https://clang.llvm.org/` (2020)
56. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: ACSAC07 (2007)
57. Martignoni, L., Paleari, R., Fresi Roglia, G., Bruschi, D.: Testing CPU emulators. In: Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA). pp. 261–272. ACM, Chicago, Illinois, USA (2009)
58. Martignoni, L., Paleari, R., Fresi Roglia, G., Bruschi, D.: Testing system virtual machines. In: Proceedings of the 2010 International Symposium on Testing and Analysis (ISSTA). Trento, Italy (2010)
59. Miramirkhani, N., Appini, M.P., Nikiforakis, N., Polychronakis, M.: Spotless sandboxes: Evading malware analysis systems using wear-and-tear artifacts. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 1009–1024 (May 2017). https://doi.org/10.1109/SP.2017.42
60. Moser, A., Krügel, C., Kirda, E.: Exploring multiple execution paths for malware analysis. In: 2007 IEEE Symposium on Security and Privacy (S&P 2007), 20-23 May 2007, Oakland, California, USA. pp. 231–245. IEEE Computer Society (2007). https://doi.org/10.1109/SP.2007.17, `https://doi.org/10.1109/SP.2007.17`

61. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system," http://bitcoin.org/bitcoin.pdf
62. Nappa, A., Xu, Z., Rafique, M.Z., Caballero, J., Gu, G.: Cyberprobe: Towards internet-scale active detection of malicious servers. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS'14) (February 2014)
63. Oprişa, C., Ignat, N.: A measure of similarity for binary programs with a hierarchical structure. In: 2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP). pp. 117–123 (2015). https://doi.org/10.1109/ICCP.2015.7312615
64. Oreans: Advanced windows software protection system. https://www.oreans.com/themida.php (2020)
65. organization, T.B.: Boost c++ libraries. https://www.boost.org/ (2020)
66. Ozarslan, S.: Online malware sandboxes. www.medium.com/@su13ym4n/15-online-sandboxes-for-malware-analysis-f8885ecb8a35 (2016)
67. Paleari, R., Martignoni, L., Roglia, G.F., Bruschi, D.: A fistful of red-pills: How to automatically generate procedures to detect cpu emulators. In: Proceedings of the 3rd USENIX Conference on Offensive Technologies. p. 2. WOOT'09, USENIX Association, USA (2009)
68. Protocol Labs: Filecoin: a decentralized storage network. https://filecoin.io/ (2020)
69. Red Hat Inc.: Ansible it automation. https://github.com/ansible (2020)
70. Rutkowska, J.: Red pill ... or how to detect VMM using (almost) one CPU instruction. https://securiteam.com/securityreviews/6z00h20bqs/ (2004)
71. Sharif, M., Lanzi, A., Giffin, J., Lee, W.: Automatic reverse engineering of malware emulators. Security and Privacy, IEEE Symposium on **0**, 94–109 (2009). https://doi.org/http://doi.ieeecomputersociety.org/10.1109/SP.2009.27
72. Tanabe, R., Ueno, W., Ishii, K., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Rossow, C.: Evasive malware via identifier implanting. In: Giuffrida, C., Bardin, S., Blanc, G. (eds.) Detection of Intrusions and Malware, and Vulnerability Assessment. pp. 162–184. Springer International Publishing, Cham (2018)
73. Tromp, J.: Cuckoo cycle: a memory bound graph-theoretic proof-of-work. In: International Conference on Financial Cryptography and Data Security. pp. 49–62. Springer (2015)
74. Tuwiner, J.: Bitmain antminer s9 review. https://www.buybitcoinworldwide.com/mining/hardware/antminer-s9/ (2017)
75. Ugarte-Pedrero, X., Balzarotti, D., Santos, I., Bringas, P.G.: Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers. In: 2015 IEEE Symposium on Security and Privacy. pp. 659–673 (May 2015). https://doi.org/10.1109/SP.2015.46
76. VirusShare: Virusshare.com - because sharing is caring. https://virusshare.com/l (2020)
77. Wang, T., Wei, T., Gu, G., Zou, W.: Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In: Proceedings of the 31st IEEE Symposium on Security and Privacy (Oakland'10) (May 2010)
78. Wikipedia: Wannacry ransomware hits prevalently windows. https://en.wikipedia.org/wiki/WannaCry_ransomware_attack/ (2017)
79. Wong, D.: Np complexity. https://www.cryptologie.net/article/43/np-complexity/ (2013)
80. Xu, Z., Nappa, A., Baykov, R., Yang, G., Caballero, J., Gu, G.: AutoProbe: Towards Automatic Active Malicious Server Probing Using Dynamic Binary Analysis. In: Proceedings of the 21st ACM Conference on Computer and Communication Security (2014)
81. Yokoyama, A., Ishii, K., Tanabe, R., Papa, Y., Yoshioka, K., Matsumoto, T., Kasama, T., Inoue, D., Brengel, M., Backes, M., Rossow, C.: Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) Research in Attacks, Intrusions, and Defenses. Springer International Publishing (2016)