A Cross-stack Approach Towards Defending Against Cryptojacking

Nada Lachtar, *Member, IEEE*, Abdulrahman Abu Elkhail, Anys Bacha *Member, IEEE*, and Hafiz Malik, *Member, IEEE*

Abstract—Cryptocurrenices are revolutionizing the way we conduct every day business. Unfortunately, cybercriminals have harnessed this technology for making profit through cryptojacking, the act of maliciously appropriating computational resources for mining cryptocurrencies. In this work, we explore a general solution for detecting cryptojacking attacks irrespective of the application type. We propose an end-to-end detection solution that leverages lightweight microarchitectural changes designed to track instructions that are commonly used in hash algorithms. An evaluation of our implementation shows negligible performance overhead while testing across a mix of workloads from the SPEC 2006 benchmarks.

Index Terms—Cryptojacking, Cryptocurrency Mining, Malware, Security.

1 Introduction

Cybercriminals are shifting their efforts towards a less risky, yet lucrative practice that is known as cryptojacking. Cryptojacking is the act of stealing execution cycles from compute resources for the purpose of mining cryptocurrencies. Several techniques have been reported for transparently appropriating execution cycles from victims. Such methods include injecting JavaScript code into high-traffic websites through common cross-site scripting attacks (XSS), forking popular projects on Github and augmenting them with cryptojacking code, and introducing seemingly benign mobile apps onto Google's Play Store that are designed to mine cryptocurrencies.

In this paper, we present a new mechanism for dynamically detecting cryptojacking attacks. Unlike previous approaches that are limited to detecting JavaScript-based cryptojacking activity within web browsers, our solution is general and can detect such malicious behavior irrespective of the application type. A key observation made in this work is that tracking a limited set of native instructions that are commonly employed in cryptographic hash functions is sufficient to reliably distinguish between benign and malicious activities with high accuracy. Starting from this observation, we propose a low overhead, end-to-end cryptojacking detection solution that spans the microarchitecture and operating system layers. We introduce lightweight microarchitectural changes that track the relevant instructions executed within the processor's pipeline. We implement OS changes to consume this information and periodically monitor the number of tagged instructions for each scheduled process through a counter. We evaluate the robustness of this approach by extensively testing real user applications and workloads from the SPEC CPU2006 suite.

Overall, this paper makes the following contributions:

- Presents a low overhead, cross-stack solution that effectively detects cryptojacking activity irrespective of the application type.
- Makes the observation that tracking a limited number of native instructions commonly employed in cryptographic hash functions is sufficient for reliably detecting cryptojacking behavior.
- Characterizes a set of instructions across a variety of applications and evaluates their suitability for cryptojacking detection.

- *The authors are with the University of Michigan, Dearborn, MI, 48128.*

2 Background and Related Work

A fundamental technology that governs cryptocurrencies is known as blockchain. Blockchain is an ordered set of blocks that are chained together to form a distributed ledger [1]. Each block is identified by a hash and contains multiple transactions for sending and receiving currency. Transactions are grouped into blocks through peer-to-peer compute nodes that are known as miners. Miners serve the purpose of validating incoming transactions and adding them to the blockchain in return for a reward. However, before a miner can add a new block, it must compete with other miners on the network and be the first to solve a complex mathematical problem that involves a significant amount of hashing. The solution to this problem is known as proof-of-work (PoW). Although other mechanisms exist, various cryptocurrencies including bitcoin rely on this PoW approach. However, a downside to popular currencies such as bitcoin is that they do not provide strong privacy guarantees. This limitation led to the rise of alternate cryptocurrencies that strive to promote complete anonymity of their transactions.

Unfortunately, the anonymous nature of these currencies has become a major attraction for cybercriminals for making profit through the process of cryptojacking. Cryptojacking is achieved by having a victim's machine mine for cryptocurrencies for the benefit of the attacker. Anonymous cryptocurrencies such as Monero and Zcash leverage

cryptographic hash algorithms that are composed of logical functions that use a combination of fundamental operations that include n-bit right rotation (R^n), n-bit right shift (S^n), exclusive or ($\oplus$), and ($\wedge$), or ($\vee$), and addition ($+$). These operations are prevalent in the SHA-3 and SHA2 algorithms that are core to both Monero and Zcash, respectively.

Prior work [2], [3] explored detection mechanisms for mitigating the effects of cryptojacking. Hong et al. [2] proposed a solution that tracks commonly used hash libraries offered by browsers. Other work [3] proposed the use of web assembly and machine learning to detect hashing activity within a browser. All of this work focused on detecting cryptojacking activity within the browser. Our work, is generic and application agnostic. Work by Demme et al. [4] explored the feasibility of using standard performance counters for malware detection. However, unlike our work, [4] lacks the ability to accurately capture counter events for individual processes which can lead to false negatives as programs are context switched on the system. In addition, such solutions are vulnerable to code obfuscation attacks [5] and generally offer low detection rates for individual classes of malware. Thus, underscoring the need for designs that can reliably fingerprint and detect non-traditional malware. Other work [6] investigated a microarchitecture driven approach with ensemble learning. However, in addition to the hardware complexity and overhead, the solution lacks the ability to detect multi-threaded malware, an evasion technique commonly employed by cryptojacking programs on multi-core systems. This is because [6] relies on core-level detectors that make their classification decisions independently. Unlike prior work, we propose a simple and low overhead approach that accurately detects cryptojacking activity while maintaining a low false positive rate. Furthermore, our design is resilient to multi-threaded, code obfuscation, and throttling attacks. Finally, this study evaluates an emerging and important class of malware that to our knowledge hasn't been explored in prior hardware-based solutions.

3 The Cryptojacking Defense System

3.1 Hardware Layer

The design leverages lightweight microarchitectural changes that enable cores to track a set of instructions that are commonly present in cryptographic hash algorithms as they are executed. This requires augmenting the front-end and the out-of-order execution modules. An overview of our cryptojacking detection system is outlined in Figure 1.

Front-end Module. The first module within our design entails the front-end. This entity serves the role of tagging a select set of instructions. To this end, we augment the decode stage with logic that tags the fetched instructions that are relevant to cryprojacking detection. Because of the hash-focused nature of cryptojacking programs, such instructions primarily span rotation, shift, and exclusive or operations. We refer to this group as `RSX` instructions. Our design tags these instructions during the decode stage of the pipeline. To facilitate in-field updates, our design allows for a programmable set of instructions to be tagged through the use of microcode that can be upgraded via firmware. Once instructions have been tagged, they are sent to the out-of-order execution engine. This transition is illustrated as step 1 in Figure 1.

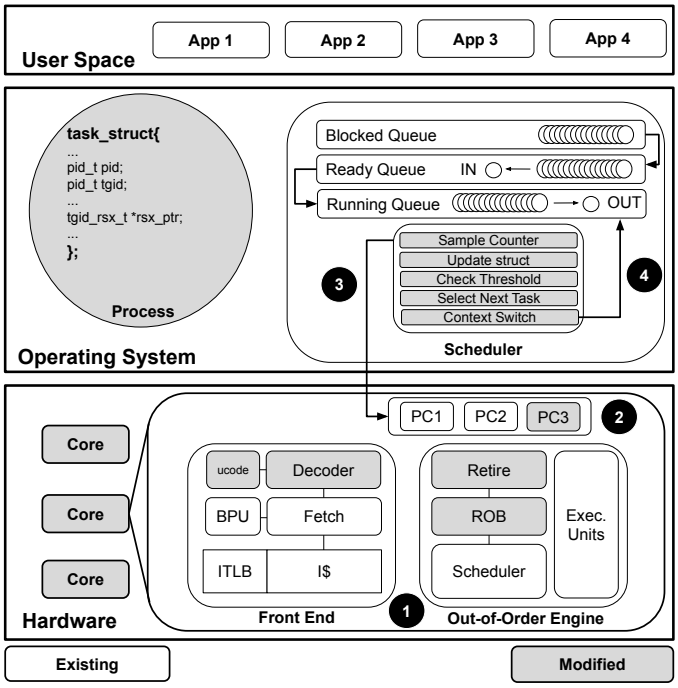


Fig. 1: Overview of the cryptojacking defense system.

Out-of-order Execution Engine. The next phase of our detection process involves the out-of-order execution engine. Once an instruction is received from the front-end module, a new entry is created for it within the re-order buffer (ROB) to preserve its sequence within the original program order. Our design makes use of an additional `RSX` bit that is added to each ROB entry. This bit is used to track any hash-related instructions that were previously tagged during the decode stage in step 1. Completed instructions continue to progress through the ROB until they reach the commit point. If an entry that has both its `RSX` and completion bits set reaches the commit point in the ROB, the retirement logic updates the performance counter to reflect that a newly `RSX` instruction has been retired. At this point, the architectural state is made visible. This step is illustrated as step 2 in Figure 1.

3.2 Operating System Layer

Our design leverages the OS scheduler to collect information from the hardware and make the necessary decisions on cryptojacking activity. To this end, the scheduler is tasked with performing a set of routine checks upon every context switch of a running process including the sampling of counters that track the number of retired `RSX` instructions. This step is shown in Figure 1 as step 3. To reduce the complexity of the hardware, we employ a single counter that aggregates number of executed `RSX` instructions. The scheduler logs the aggregated value into an `rsx_count` field that is shown in Listing 1. The `rsx_count` is in turn accessible from the `task_struct` of the running process through the `rsx_ptr` field outlined in Figure 1. Once the aforementioned information is recorded, the scheduler proceeds to run the next queued task. We monitor each process

over a predefined period to ensure that it has executed a sustained rate of RSX instructions before a decision is made. An alert is sent to the user in the event that the threshold is exceeded. This is illustrated as step ❹ in Figure 1. The threshold and the monitoring period are controlled through a set of kernel tunables that can be updated at runtime through the /proc virtual file system.

Our system supports the detection of multi-threaded cryptojacking services. This is achieved by aggregating the overall count of RSX instructions across threads that belong to the same group based on their thread group ID (tgid). These threads share a common structure (tgid_rsx_t) that contains the RSX count (rsx_count) and the number of threads referencing the structure (tcount). Whenever tcount is reduced to zero, the structure is deallocated implying that all of the threads have been terminated.

```
struct tgid_rsx_t {
    refcount_t rsx_count;
    refcount_t tcount;
};
```

Listing 1: Structure for tracking RSX instructions.

4 Threat Model

We assume no privilege escalations have occurred on the system and that an attacker can distribute mining activity using multiple threads to avoid detection. We focus on tracking instructions that can lead to profitable cryptojacking attacks. Therefore, we assume an attacker limits obfuscation attacks to instruction substitutions that yield relatively high throughput. For instance, an attacker could substitute rotation instructions with different shift operations to evade detection and vice versa while still being able to attain relatively high hash rates. On the other hand, substituting an XOR instruction with ADD instructions and a series of bitwise operations would be considered uneconomical for cryptojacking purposes because of the impact to throughput. While our solution could be programmed to track various instruction types, we do not focus on substitution attacks that can render obfuscated malware ineffective at mining.

5 Evaluation

5.1 Methodology

We conducted experiments using a 4-core, out-of-order x86 processor that we modeled using the gem5 simulator. The parameters of the modeled hardware are summarized in Table 1. In addition, we used the Ubuntu 16.04 OS with modifications to the Linux v4.19.91 kernel to support our solution. We characterized the frequency of x86 instructions across the SPEC CPU2006 benchmark suite, SHA-2, SHA-3, and AES workloads over a period of 1-billion instructions. Furthermore, we tested 125 real user applications that span productivity programs such as Office suite, Google Chrome, and Eclipse that were all configured to run on Linux. We ran common web services such as YouTube, Amazon, and ESPN while using the Google Chrome browser. To efficiently utilize the user interfaces of the applications while tracking the instructions they execute, we used Intel's

Hardware Configuration	
Cores	4 (out-of-order)
ISA	x86
Frequency	2.0GHz
IL1/DL1 Size	32KB
IL1/DL1 Block Size	64B
IL1/DL1 Associativity	8-way
IL1/DL1 Latency	2 cycles
Coherence Protocol	MESI
L2 Size	2MB
L2 Block Size	64B
L2 Associativity	16-way
L2 Latency	20 cycles
Memory Type	DDR4-2400 SDRAM
Memory Size	3GB

TABLE 1: Summary of hardware configurations.

Software Development Emulator (SDE) [7]. Each application was used interactively through SDE. In addition to the aforementioned user applications, we tested Monero and Zcash by having them mine cryptocurrencies on live testnets as we recorded their instructions.

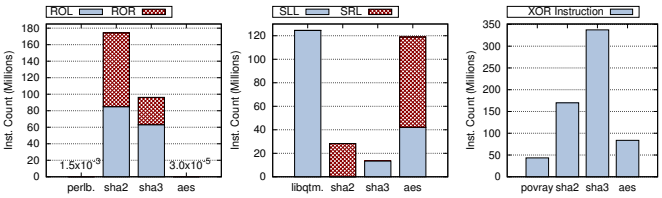
5.2 Analysis

We characterized the RSX instructions present in SPEC2K6, SHA-2, SHA-3, and AES in order to determine the feasibility of our design tracking such instructions. Instruction counts were recorded after the execution of 1 billion instructions.

Figure 2a shows the count for the rotation instructions ROR and ROL. Although the SHA-2 and SHA-3 algorithms employ right rotation operations, the compiler utilizes instructions in both directions. We observe that the count of ROR instructions is 89M (million) and 33M for SHA-2 and SHA-3, respectively. On the other hand, the remaining workloads have zero ROR instructions with the exception of the following workloads which have very low occurrences: *perlbench* (15 instructions), *omnetpp* (3 instructions), and AES (3 instructions). We observe a similar trend with ROL. We find that the count of the ROL instruction is 85M and 63M for SHA-2 and SHA-3, respectively. However, the remaining workloads have an average of 85 ROR occurrences ranging between 4 to 2K instructions. This data suggests that observing a large number of ROR and ROL instructions during execution is a strong indicator for cryptojacking activity.

Figure 2b shows that although SHA-3 doesn't directly consume right shift instructions (SRL), SRL is central to SHA-2. On average, SHA-2 exhibits 28M SRL instructions. This is 10x the instruction count relative to the benchmarks in SPEC2K6. On the other hand, the AES algorithm shows a significantly higher count (76M) which is 2.7x higher relative to the SRL instructions observed in SHA-2. We also find that the left shift instruction (SLL) count is relatively low in cryptographic functions. For instance, we observe that *libquantum* has 124M SLL operations. This is 3x and 9x the count relative to AES and SHA-2, respectively. In general, shift instructions on their own are not a good feature.

We observe higher counts for the XOR instruction relative to the rest of the workloads. This is summarized in Figure 2c. We find 170M and 337M XOR instructions in SHA-2 and SHA3, respectively. On the other hand, XOR instructions in the SPEC2K6 benchmarks range between 0.1M (*libquantum*) and 43M (*povray*). In comparison, the count in SHA-2 and



(a) Rotate (b) Shift (c) Exclusive OR

Fig. 2: Breakdown of (a) rotate, (b) shift, and (c) exclusive or instructions in SHA-2, SHA-3, AES, and SPEC2K6 benchmark with highest count over a 1B instruction window.

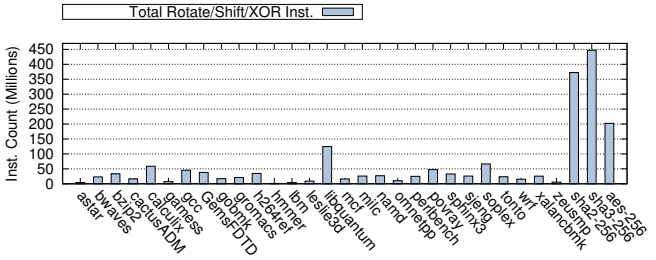


Fig. 3: Cumulative number of Rotate, Shift, and Exclusive OR (RSX) instructions after executing 1 billion instructions.

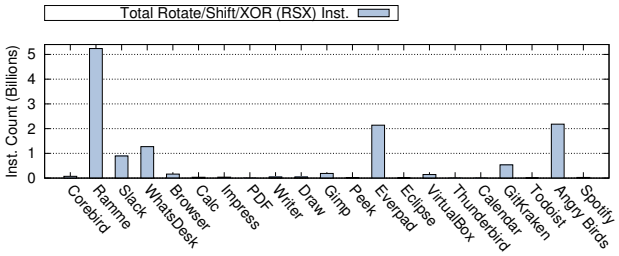
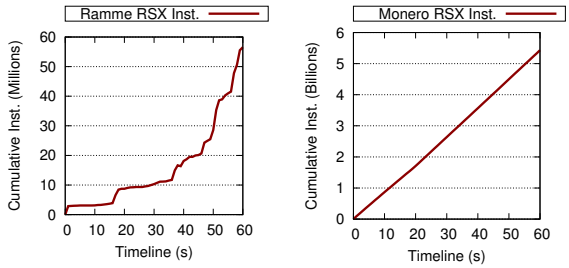


Fig. 4: Cumulative number of RSX instructions in real user applications after executing for one hour.



(a) Ramme (b) Monero

Fig. 5: Cumulative RSX instructions as a function of time over a one minute period for (a) Ramme and (b) Monero.

SHA-3 is 12x and 23x higher relative to the average workload in SPEC2K6, and 2x and 4x relative to AES.

Our results show that rotation instructions are sufficient to detect cryptojacking activity. However, an attacker could subvert a rotation-based detection by using a combination of shift instructions and still maintain a reasonably high hash rate. For instance, an n-bit rotation in the left direction (R_l^n) could be replaced by a series of n-bit left shift (S_l^n), m-bit right shift (S_r^m), and or ($\vee$) instructions. Therefore, in addition to keeping the design simple, our solution tracks the cumulative count of all `RSX` instructions to mitigate code obfuscation attacks. Figure 3 shows the cumulative number of `RSX` instructions after executing the benchmarks for 1 billion instructions. We observe that SHA-2 and SHA-3 have 3x and 3.5x the amount of instructions compared to the *libquantum* benchmark. Although *libquantum* has the highest number of `RSX` instructions in the SPEC2K6 suite due to 124M `SLL` operations, the count is significantly less than that of SHA-2 and SHA-3.

Figure 4 shows the `RSX` count across real applications that we tested over a period of one hour. On average, workloads have 0.9B `RSX` instructions with Ramme having the highest count. Ramme, a social media app equivalent to Instagram, had a total of 5.3B `RSX` instructions. A closer look into Ramme reveals that over 77% of the `RSX` instructions were shift operations and 20% of the remaining `RSX` instructions were `XOR`. To understand the effectiveness of using `RSX` instructions for cryptojacking detection, we compared Ramme to Monero and Zcash over the same one hour execution period. Overall, Monero and Zcash had significantly higher `RSX` counts relative to the user applications shown in Figure 4. Monero had 342B `RSX` instructions which is more than 65x the `RSX` count relative to Ramme. Zcash had a far greater `RSX` count of 3×10^3B compared to Ramme. Figure 5 shows the cumulative `RSX` count over a one minute period. We find that Monero's `RSX` count is at least two orders of magnitude higher than that of Ramme. After testing several applications, we determined that using a threshold of 2.5B `RSX` inst./min allows us to detect the Monero and Zcash workloads with true and false positive rates of 100% and under 2%, respectively. We observe that the false positives only occur when continuously running the core cryptographic functions SHA-2, SHA-3, and AES. Furthermore, an attacker may throttle the execution of cryptojacking malware. Our solution can sustain the aforementioned rates even when an attacker throttles execution by more than 50%.

5.3 Performance Overhead

Our solution incurs insignificant overhead. All of the SPEC2K6 workloads exhibit less than 1% overhead.

6 Conclusion

In this work, we present a cross-stack solution for defending against cryptojacking. We evaluate a simple, yet effective approach that adds minimal overhead to the overall platform.

Acknowledgment

This work was supported in part by the National Science Foundation under grant CNS-1947580.

References

[1] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies*. " O'Reilly Media, Inc.", 2014.

[2] G. Hong, Z. Yang, S. Yang, L. Zhang, Y. Nan, Z. Zhang, M. Yang, Y. Zhang, Z. Qian, and H. Duan, "How you get shot in the back: A systematical study about cryptojacking in the real world," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1701–1713, 2018.

[3] A. Kharraz, Z. Ma, P. Murley, C. Lever, J. Mason, A. Miller, N. Borisov, M. Antonakakis, and M. Bailey, "Outguard: Detecting in-browser covert cryptocurrency mining in the wild," in *The World Wide Web Conference*, pp. 840–852, 2019.
[4] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 559–570, 2013.
[5] B. Zhou, A. Gupta, R. Jahanshahi, M. Egele, and A. Joshi, "Hardware performance counters can detect malware: Myth or fact?," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security (Asia CCS)*, pp. 457–468, ACM, 2018.
[6] K. N. Khasawneh, M. Ozsoy, C. Donovick, N. Abu-Ghazaleh, and D. Ponomarev, "Ensemble learning for low-level hardware-supported malware detection," in *International Symposium on Recent Advances in Intrusion Detection*, pp. 3–25, Springer, 2015.
[7] A. Tal, "Intel software development emulator," 2020.