An Application Agnostic Defense Against the Dark Arts of Cryptojacking

Nada Lachtar, Abdulrahman Abu Elkhail, Anys Bacha, Hafiz Malik
University of Michigan, Dearborn, USA
{nlachtar, abdkhail, bacha, hafiz}@umich.edu

***Abstract*—The popularity of cryptocurrencies has garnered interest from cybercriminals, spurring an onslaught of cryptojacking campaigns that aim to hijack computational resources for the purpose of mining cryptocurrencies. In this paper, we present a cross-stack cryptojacking defense system that spans the hardware and OS layers. Unlike prior work that is confined to detecting cryptojacking behavior within web browsers, our solution is application agnostic. We show that tracking instructions that are frequently used in cryptographic hash functions serve as reliable signatures for fingerprinting cryptojacking activity. We demonstrate that our solution is resilient to multi-threaded and throttling evasion techniques that are commonly employed by cryptojacking malware. We characterize the robustness of our solution by extensively testing a diverse set of workloads that include real consumer applications. Finally, an evaluation of our proof-of-concept implementation shows minimal performance impact while running a mix of benchmark applications.**

I. Introduction

From startup investments to dining out across the street, cryptocurrencies are revolutionizing the way we conduct every day business. It was estimated in 2019 that over 11 million cryptocurrency transactions occurred on a daily basis [34], prompting various governments to launch their own state-backed cryptocurrencies as a way of boosting their economies [15]. Unfortunately, the popularity of cryptocurrencies has garnered interest from cybercriminals, spurring an onslaught of creative attacks on computer systems in order to harness this technology for profit.

Cybercriminals are constantly seeking ways to make their attacks more profitable. For instance, several governments have fallen victim to a slew of ransomware attacks within recent years. This resulted in the aforementioned entities being extorted for ransom in return for restoring their maliciously encrypted data. Although ransomware has served cybercriminals well in the past, attackers are resorting to cryptojacking, a less invasive and more lucrative form of malware.

Cryptojacking involves appropriating computational resources from a victim for the purpose of mining cryptocurrencies. Although cryptojacking is considered a low risk practice within the cybercriminal world, attackers can earn more than $1.7 million for 10 million victims per month [17]. As a result, cryptojacking has burgeoned aggressively across a multitude of platforms that span mobile devices, PCs, network infrastructure, and servers, with several campaigns having notable success [6], [10], [11], [26]. For instance, the Smominru cryptojacking campaign [26] was estimated to have hijacked half of a million PCs for mining cryptocurrencies. Another cybersecurity firm discovered that an average of 3,000 Microsoft SQL servers were being infected every day for running mining scripts [10]. Similarly, supercomputers across Europe that are often tasked with solving important global and computationally complex problems were hijacked for mining cryptocurrencies [11]. Such campaigns can have adverse effects on businesses and consumers. This includes the unavailability of compute resources, as well as, high energy costs due to systems being enslaved to cryptojacking malware.

In addition to higher energy costs and hijacked computational resources that can lead to denial of service, cryptojacking can have adverse effects on the reliability of systems. The deep submicron transistors that serve as the fundamental building block in today's processors are vulnerable to circuit aging effects that occur in sustained high power execution environments. Such effects are induced by Negative Bias Temperature Instability (NBTI) and Hot-Carrier Injection (HCI) phenomena [19], [33], [37] which can permanently damage a given device. To this end, researchers have reported laptops being damaged after falling victim to Monero-based cryptocurrency miners, as well as, casings of smartphones being melted [9], [29].

Various approaches have been found in the wild for luring victims into cryptojacking practices. This includes compromising commonly visited websites and infecting them with malicious JavaScript code [2], forking popular Github projects and injecting into them malicious code [4] that could propagate to cloud-based containers and in turn exhaust available compute credits, as well as disguising cryptojacking malware as legitimate apps onto Google Play and Microsoft Stores [18], [39]. All of the aforementioned techniques share the common goal of appropriating execution cycles from the victim's machine with the intent of mining cryptocurrencies.

This paper proposes a novel solution for dynamically detecting cryptojacking attacks. Unlike prior work that has focused on detecting cryptojacking activity within web browsers [5], [17], [20], [21], [38], our mechanism is more general, and therefore, application agnostic. More importantly, our solution mitigates commonly used evasion techniques, such as code obfuscation, multi-threaded, and throttling attacks. We find that selecting key instructions that are frequently used in cryptographic hash functions serve as reliable signatures for fingerprinting cryptojacking behavior. We leverage these signatures to implement a low-cost, yet effective proof-of-concept that can accurately detect cryptojacking activity irrespective of the application type. We accomplish this by

harnessing innovations at the microarchitecture layer that track a programmable set of instructions as they progress through the processor. We develop the necessary modules within the operating system to periodically collect the frequency of tagged instructions from the hardware layer for all scheduled processes while maintaining low overheads. We characterize the robustness of our approach by extensively testing a wide range of applications that span multiple categories, including: social media, productivity, communication, entertainment, high performance computing, cryptography, as well as non-mining cryptocurrency and decentralized applications.

Overall, this paper makes the following contributions:

- Proposes an application agnostic design that harnesses innovations at the microarchitecture and OS layers for defending against emerging cryptojacking attacks.
- Demonstrates that monitoring instructions that are commonly present in cryptographic functions are suitable for cryptojacking detection.
- Characterizes a variety of instruction types in cryptographic functions, benchmarks, and commonly deployed user applications, while evaluating their relevance for cryptojacking detection with high accuracy.
- Presents a design that is robust against code obfuscation, multi-threaded, and throttling attacks while maintaining a low false positive rate.

The rest of this paper is organized as follows: Section II provides background and motivation information. Section III discusses the threat model. Section IV describes the design of the proposed system. Section V presents the methodology and experimental framework used in this work. Section VI discusses the results of our evaluation. Section VII details related work; and Section VIII concludes.

II. Background and Motivation

A. Out of Order Execution

Modern processors employ a variety of latency hiding techniques in order to improve execution performance. Such techniques include temporarily executing program instructions in an out of order fashion as a way of utilizing hardware resources more efficiently. With this approach, instructions are fetched into the processor in their original program order through a front-end module. This module is responsible for fetching a continuous stream of instructions from the instruction cache into the processor's pipeline. To supply a high-bandwidth of instructions into the pipeline, the front-end makes use of a branch prediction unit (BPU) that speculates the next set of instructions to be executed before branches are resolved, and fetches them for execution. Fetched instructions are decoded to determine the functional units they require and then forwarded to the out of order engine for further processing. On complex instruction set computer (CISC) architectures, such as x86, the decoder also translates incoming instructions into simpler micro-instructions.

When instructions arrive to the out-of-order execution engine, entries within the re-order buffer (ROB) and reservation station structures are allocated. The ROB is a circular queue that tracks outstanding instructions in the out-of-order execution engine along with the completion status information for each instruction. It is the entity that maintains the original sequence of instructions and ensures that completed instructions are retired in program order. This necessary ordering is maintained through the use of head and tail pointers that are updated as instructions are retired. Instructions continue to advance within the re-order buffer towards the head until they are committed and their architectural state is made visible.

The actual out-of-order execution, on the other hand, is provided through a scheduler that references reservation stations. These are primarily buffers for the actual instructions before they are dispatched to the relevant execution units. The scheduler is used to determine when the buffered instructions are eligible for dispatch to the execution units. In addition, the scheduler monitors the results of previous instructions that are generated by the execution units to determine when the dependencies of buffered instructions are resolved and can proceed to the execution units. Whenever the dependencies are met, the scheduler dispatches the appropriate instruction regardless of its original program order assuming the appropriate functional unit is available. Our solution leverages modifications to the front-end and out-of-order execution units for tracking instructions in support of cryptojacking detection.

B. Cryptocurrencies

Unlike traditional currencies that are governed through a central authority, cryptocurrencies are completely virtual and decentralized. They rely on blockchain as a core technology. Blockchain is a distributed ledger that is established through a collection of interconnected blocks [3]. Blocks within the blockchain are identified by their hashes, where each block can contain several transactions. In addition, every block includes within its header, the hash of its parent block in order to form the blockchain. Outstanding transactions are formed into blocks that are in turn processed by a network of peer-to-peer nodes, referred to as miners. Although miners fulfill the role of validating newly created blocks using a decentralized consensus mechanism, they also compete with one another over receiving a reward in return for their effort. The first miner that solves a complex mathematical puzzle is rewarded once the block is verified and deposited onto the blockchain. The solution to the puzzle is known as proof-of-work (PoW) and involves a considerable amount of hash operations. Although other approaches exist, cryptocurrencies such as Monero [25] and Zcash [40] that are designed to maintain their transactions anonymous, rely on the PoW approach. Unfortunately, the anonymity of such cryptocurrencies has garnered attention by cybercriminals, giving rise to a slew of cryptojacking attacks.

C. Secure Hash Algorithms

Hashing is a core component of cryptocurrency algorithms. These algorithms rely on a significant amount of hash operations as part of recording transactions on the public ledger while making them immutable. For instance, transactions

within a given block undergo multiple iterations of hash operations as they are progressively merged using a Merkle tree. In addition to merging transactions, several rounds of hashing are applied as part of solving the hash difficulty imposed by the algorithm.

In this study, we focus on the SHA-3 and SHA-2 secure hash algorithms which serve as core functions for most cryptocurrencies, including ones that promote anonymity, such as Monero [25] and Zcash [40]. Such functions rely on elementary operations that include addition ($+$), logical or ($\vee$), logical and ($\wedge$), exclusive or ($\oplus$), n-bit right shift (S^n), and n-bit right rotation (R^n).

Secure Hash Algorithm 3 (SHA-3). SHA-3 is a one-way cryptographic hash algorithm that is based on a sponge construction. Each block b_i within a message M undergoes a five-stage process that is denoted as θ, ρ, π, χ, and ι.

Step θ is designed to ensure diffusion. This is accomplished by manipulating a sequence of arrays, A, B, C, and D, using modulo 5 arithmetic in addition to other logical operations. The arrays are indexed using indices x and y. Equation (1a) - (1c) summarize the operations involved in step θ.

$$C_x = A_{x,0} \oplus A_{x,1} \oplus A_{x,1} \oplus A_{x,2} \oplus A_{x,3} \oplus A_{x,4} \tag{1a}$$
$$D_x = C_{x-1} \oplus R^1(C_{x+1}) \tag{1b}$$
$$A_{x,y} = A_{x,y} \oplus D_x \tag{1c}$$

Steps ρ and π are used to derive an auxiliary array B from the state array A through rotation and permutation. Both of these steps are summarized in equation (2).

$$B_{y,2x+3y} = R^{r_{x,y}}(A_{x,y}) \tag{2}$$

Step χ serves the purpose of manipulating the auxiliary array B that was previously computed in steps ρ and π. This step is defined in equation (3).

$$A_{x,y} = B_{x,y} \oplus (\bar{B}_{x+1,y} \wedge B_{x+2,y}) \tag{3}$$

The final step is ι. It combines each state array element from A with a predefined constant, RC_i in order to undo any symmetry that was produced by the previous steps. This step is illustrated in equation (4).

$$A_{0,0} = A_{0,0} \oplus RC_i \tag{4}$$

Overall, the aforementioned equations show that a large portion of the SHA-3 algorithm entails performing exclusive or and rotation operations.

Secure Hash Algorithm 2 (SHA-2). Unlike SHA-3, SHA-2 is a cryptographic hash function that relies on a Merkle-Damgard construction. Its compression function f is composed of six logical components that are executed repeatedly for each block b_i within a message M. The logical functions and the operands they use are outlined in equations (5a) - (5f).

$$Ch(x,y,z) = (x \wedge y) \oplus (\bar{x} \wedge y) \tag{5a}$$
$$Maj_0(x) = (x \wedge y) \oplus (x \wedge y) \oplus (y \wedge z) \tag{5b}$$
$$\Sigma_0(x) = R^2(x) \oplus R^{13}(x) \oplus R^{22}(x) \tag{5c}$$
$$\Sigma_1(x) = R^6(x) \oplus R^{11}(x) \oplus R^{25}(x) \tag{5d}$$
$$\sigma_0(x) = R^7(x) \oplus R^{18}(x) \oplus S^3(x) \tag{5e}$$
$$\sigma_1(x) = R^{17}(x) \oplus R^{19}(x) \oplus S^{10}(x) \tag{5f}$$

Despite the differences between the SHA-2 and SHA-3 algorithms, SHA-2 also requires performing a large amount of exclusive or and rotation operations. Furthermore, it is important to note that in addition to the use of the aforementioned secure hash algorithms, anonymous cryptocurrencies, such as Monero and Zcash, incorporate the Advanced Encryption Standard (AES) [1] and the BLAKE2 hash algorithm [8], [32]. These algorithms also rely on n-bit rotation (R^n), n-bit shift (S^n), exclusive or ($\oplus$) operations, and several other arithmetic operations enclosed in a memory hard loop. As such, our study considers the aforementioned operations in the context of other cryptographic algorithms that utilize the same set of operations.

D. Hash Operations in Mining Services

To understand the amount of hashing anonymous cryptocurrencies require and their respective operations, we conducted a set of experiments using one of the most popular anonymous cryptocurrencies, Monero. We began by statically analyzing the core function that performs the SHA-3 hashing (Keccak) within Monero's CryptoNight algorithm [12]. For instance, the aforementioned function is the entity that is responsible for carrying out the previously described five stages of the sponge construction: θ, ρ, π, χ, and ι. To this end, we disassembled a Monero binary using the GNU `objdump` utility and examined its compiled instructions. More specifically, we analyzed the x86 assembly instructions present in the `keccakf()` function. Figure 1 summarizes the distribution of the compiled instructions contained in this core function of the CryptoNight algorithm. We observe that more than half of the opcodes (56%) are in the form of `MOV` instructions. We also find in the code typical stack-based instructions, such as `PUSH` and `POP` that are used for accessing variables, as well as, setting up and restoring stack-frames as a result of entering and exiting the function. Most importantly, we find other instructions that are central to hash operations. These include `XOR`, which represents 24% of the overall instructions in the hash function. In addition, we find several `AND` instructions which are also employed in hashing. This represents about 8% of the overall instructions within the function. Finally, we find `ROR` and `ROL` rotation instructions which account for 2% of the total instructions. While Figure 1 doesn't take into consideration the frequency of the instructions, as a result of looping through multiple rounds, we can see that the critical instructions that are key to the sponge construction are present within the SHA-3 function of the CryptoNight algorithm.

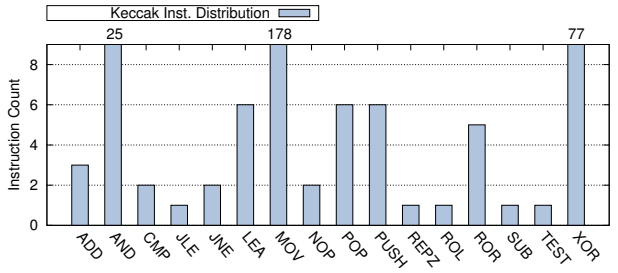


Fig. 1: Distribution of compiled instructions in the core Keccak module within Monero's CryptoNight algorithm.

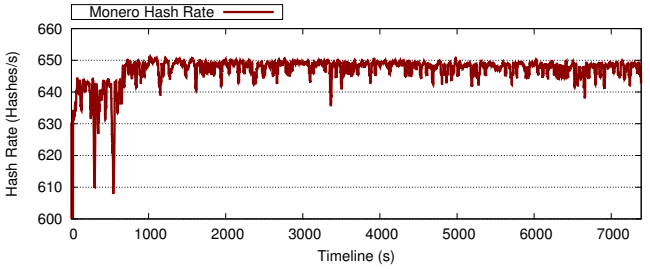


Fig. 2: Hash rate of a live Monero cryptocurrency service while mining on a Core i7 3370 based system.

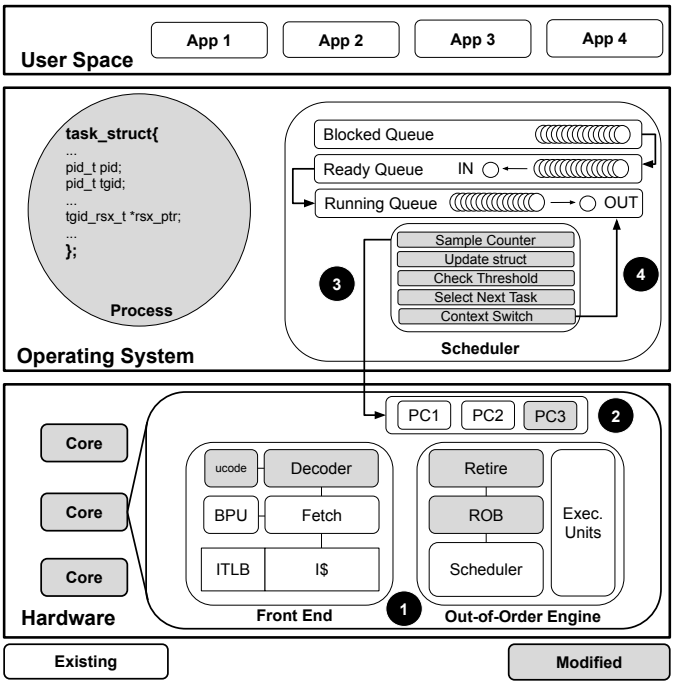


Fig. 3: Main components of the cryptojacking defense system. The components we modified are highlighted in gray.

In order to estimate the frequency of the previously described instructions within cryptocurrency services, we instrumented the code available through the Monero project [25]. We then deployed this instrumented code on a live mining service and recorded the hash rate our node observed while mining. Figure 2 illustrates the average hash rate observed by our node as a function of time. We observe that on an x86 system equipped with a 4-core Intel Core i7 3370 processor and 8GB of memory, an average of 647 Hashes/s was recorded while mining on the Monero service. We see that this rate is sustained over a period that exceeds two hours. The lowest hash rate that we recorded throughout this period was 564 Hashes/s. This experiment underscores the continuous and aggressive nature of hash operations that systems engaged in cryptocurrency mining will exhibit in a sustained fashion over extended periods.

III. Threat Model

Our solution is designed to provide application agnostic protection against cryptojacking. This includes protecting against attacks that are sourced through web browsers and standard applications. As such, we assume attackers can initiate cryptojacking activity through any of the following methods:

- **Web Browsers.** Attackers leverage a drive-by approach where victims visit a website infected with a malicious script. The mining script is launched on the victim's machine after the compromised page is visited [24].
- **Non-browser Applications.** Similar to browser extensions, attackers publish seemingly benign applications on app stores, often in the categories of gaming and education in order to lure their victims [18], [27], [39]. Other methods include forking popular Github projects and inject into them malicious code that the victims would in turn build from source [28].

In addition to the above, our design assumes that the attack is confined to user space and that the system is free from any privilege escalations. We also assume that mining could be performed across multiple threads in a throttled fashion in order to evade detection. We assume obfuscation attacks are confined to instruction substitutions that result in reasonably high hash rates. For example, an attacker could use shift instructions in lieu of rotate instructions to avoid detection while remaining profitable. However, substituting an exclusive or instruction with a sequence of bitwise operations would be considered uneconomical from the attacker's point of view since such obfuscation techniques would render cryptojacking ineffective at mining.

IV. The Cryptojacking Defense System

Our defense system employs a cross-stack design that spans the hardware and OS layers for detecting cryptojacking activity. An overview of this system is depicted in Figure 3.

A. Hardware Layer

A key observation to our approach involves monitoring instructions that are frequently executed in hash functions as a way of detecting malicious behavior. Consequently, we enhance the microarchitecture of the front-end and out-of-order execution components belonging to the CPU cores to enable autonomous tracking of pre-programmed instructions.

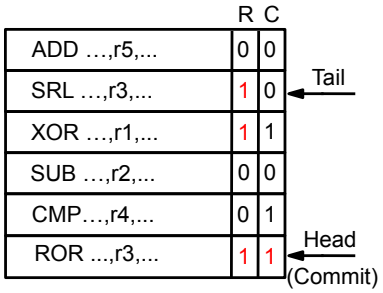


Fig. 4: Example showing the use of the RSX bit for tracking crypto-related instructions within the re-order buffer (ROB).

Front-end Module. This entity is responsible for tracking the relevant instructions associated with cryptojacking malware. As such, fetched instructions that are deemed potentially malicious are tagged through logic that we embed in the pipeline's decoder. In our design, we tag, rotate, shift, and exclusive or operations (RSX) since these by in large encompass the main set of instructions hash functions rely on, and are in turn core to cryptojacking malware. Although this work focuses on RSX instructions, our solution is designed to be field upgradable. Consequently, our design incorporates microcode to allow reprogramming of the instructions to be tagged. Such updates can be transparently initiated from the OS in the form of firmware updates, which makes our solution scalable to future malware attacks.

Out-of-order Execution Engine. This component is responsible for tracking the instructions tagged during the decode stage as they undergo out of order execution. To this end, an entry within the ROB is allocated whenever a tagged instruction is received from the front-end component. Such entries are used to maintain the original sequence of each instruction produced by the compiler. Each entry in the re-order buffer is augmented with a special bit that is used to mark incoming RSX instructions that were tagged during the decode stage. Figure 4 shows an example of how the RSX bit (denoted as "R") is used for tracking crypto-related instructions within the ROB. We can see that the relevant instructions, such as SRL, XOR, and ROR have their RSX bits set. Completed instructions that are denoted with a status bit "C" in Figure 4 continue to advance towards the commit point. Once an entry reaches the commit point, the retirement logic examines the status of the "R" and "C" bits. If both bits are set, the design updates the performance counter to indicate that a new RSX instruction has retired, as shown in step ❷ in Figure 3.

B. Operating System Layer

An important component of our solution lies within the OS layer. Our design leverages the OS scheduler to collect the necessary information from the hardware. We choose using the scheduler for this purpose primarily to enable detecting cryptojacking activity at the process level. As a result, we design the scheduler to perform various housekeeping tasks every time a running process is context switched. Such tasks encompass counter sampling, data structures updates, and RSX threshold checking. This step is illustrated as step ❸ in Figure 3.

In order to keep the hardware design low-cost, we employ a single counter for tracking the aggregate number of crypto-related instructions as they are executed through the processor's pipeline. The scheduler uses the rsx_count field, shown in Listing 1, for tracking the cumulative number of RSX instructions the running process has executed. This field is referenced by the scheduler through an rsx_ptr field that is located in the overall task_struct data structure of the running process, as illustrated in Figure 3. Once the necessary RSX information has been recorded, the scheduler proceeds to run the next available task in the ready queue. Each process is monitored over a pre-determined execution period (e.g. one minute) that goes beyond the scheduler's time slice. This is performed to ensure that no premature actions are taken unless the scheduler observes a continuous stream of RSX instructions. The user receives an alert once the count of RSX instructions surpasses a pre-defined threshold. This is shown as step ❹ in Figure 3. This approach allows us to reduce the potential of false positives as a result of programs that may experience short-lived peaks in the number of executed RSX instructions. In our design, the monitoring period and threshold for a process are dynamically programmable at runtime using kernel tunables that can be updated using procfs. Finally, to reduce the overhead of our detection system, our solution limits its monitoring to non-root processes. We achieve this by having the scheduler check for a non-zero uid before performing any additional processing.

An important aspect of our design is concerned with the detection of cryptojacking malware that distributes mining across multiple threads to evade detection. To this end, our system is designed to aggregate the count of rotate, shift, and exclusive or instructions across launched threads that belong to the same program. Although all threads created in Linux will possess unique IDs, threads that are created from a single process will share a common thread group ID (tgid). Based on this, we maintain a shared structure (tgid_rsx_t) across all the threads that belong to the same thread group (share the same tgid). This shared structure is accessed within the scheduler through the rsx_ptr pointer that is maintained within the task_struct of each thread. The tgid_rsx_t structure contains semaphored counters that are used to track the count of the RSX instructions (rsx_count), as well as the number of active threads referencing the structure (tcount).

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

Listing 1: Data Structure for tracking RSX instructions.

The sharing of the tgid_rsx_t structure across multiple threads is achieved through modifications in the kernel's _do_fork() routine that is used by the clone() system

call. Whenever a light-weight process is created and before it is activated, we examine the `tgid`. If the new light weight process has the same `tgid` as its parent, then we simply set the `rsx_ptr` field in the `task_struct` to point to the parent's existing `tgid_rsx_t` structure. However, if the parent and the child have different `tgid`, a new `tgid_rsx_t` is allocated and initialized. Listing 2 summarizes the logic for sharing this structure across light-weight processes within a multi-threaded program. Finally, the code makes use of counters in the structure that are atomically referenced. The `tcount` field is used to track the count of threads actively using the structure. The structure is freed once `tcount` reaches zero since this indicates that all the threads have been terminated. The `rsx_count` field, on the other hand, is used for tracking the overall number of `RSX` instructions.

```
long _do_fork(struct kernel_clone_args
    *args) {
  ...
  //Check if parent and child share same
     tgid
  if(p->tgid != parent->tgid) {
     create_tgid_crypt_struct(p);
  }
  else {
     p->rsx_ptr = parent->rsx_ptr;
  }
  wake_up_new_task(p);
  ...
}
```

Listing 2: Code sample for sharing the `tgid_rsx_t` structure across threads that belong to the same thread group (`tgid`).

V. Methodology

We conducted experiments using the gem5 cycle-accurate simulator [7]. We modeled a 4-core, out-of-order x86 processor using the simulator in Full System mode integrated with our solution. The configuration of the architectural parameters we modeled are listed in Table I. In addition, we modified the Linux v4.19.91 kernel that was configured to run the Ubuntu 16.04 OS distribution. Furthermore, we characterized the frequency of different instructions from the x86 ISA across a variety of compute-bound and memory-bound workloads present in the SPEC CPU2K6 benchmark suite [16], as well as cryptographic functions such as SHA-2, SHA-3, and AES. Each workload was characterized for a duration of 1 billion instructions for comparison. We used the SPEC2K6 benchmark suite to evaluate the performance overhead of our solution. For characterization purposes, we enabled gem5 to use multiple performance counters that tracked a variety of x86 instruction types separately while we tested our benchmarks.

To ensure coverage of our solution under different application requirements, we tested more than 150 real user applications independently for a duration of one min. In addition, we tested common applications that spanned four categories: social, communication, productivity, and entertainment, more extensively, for a duration of one hour. The aforementioned categories and their corresponding applications tested over the extended period are summarized in Table II. Since computer systems are used in the context of productivity, we tested our desktop environment (Ubuntu) against several productivity applications. We also tested non-mining applications that are cryptocurrency focused over the same one hour duration. This included testing a variety of crypto-wallets that were connected to live services, in addition to a decentralized application that was configured to interact with a smart contract. Furthermore, we evaluated Zcash and Monero workloads after dispatching them to mine on live services. Finally, we used Intel's Software Development Emulator (SDE) [36] to record the executed instructions as we interacted with each user application.

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 I: Architectural configuration parameters.

VI. Evaluation

A. Instruction Breakdown in Benchmarks

We characterized a variety of instructions in SPEC2K6, AES, in SHA-3, and SHA-2 in order to identify a reliable set of instructions to be tracked by our design. This includes shift left (`SL`), shift right (`SR`), exclusive or (`XOR`), rotate left (`RL`), and rotate right (`RR`) instructions. We focus on the aforementioned instructions since they are fundamental to the SHA-3 and SHA-2 algorithms that are employed in popular anonymous cryptocurrencies, such as Monero and Zcash.

Figure 5 shows the instruction count in millions after running each workload for 1 billion instructions. We find that the number of shift right operations (`SR`) are prevalent in SHA-2. Although SHA-3 doesn't use any `SR` operations as part of its sponge construction, `SR` is central to the Σ and σ functions

Category	Applications
Social	*Corebird (Twitter), Ramme (Instagram)*
Communication	*Slack, Skype, WhatsDesk (WhatsApp)*
Productivity	*Calc (Excel), Impress (Power Point), PDF, Writer (Word), Draw (Visio), Gimp, Peek (Screen Recorder), Everpad (Evernote), Eclipse, VirtualBox, Thunderbird, Calendar, Browser, Todoist, GitKraken*
Entertainment	*Angry Birds, Spotify*

TABLE II: Applications extensively tested over 1 hour period.

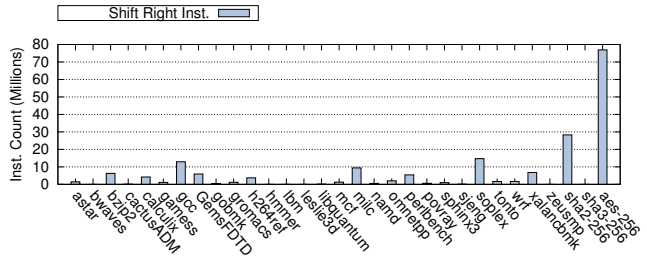


Fig. 5: Number of Shift Right (SR) instructions (in millions).

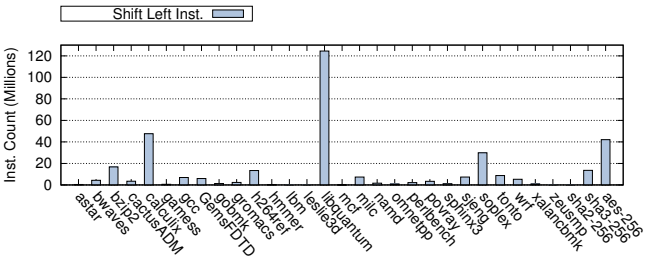


Fig. 6: Number of Shift Left (SL) instructions (in millions).

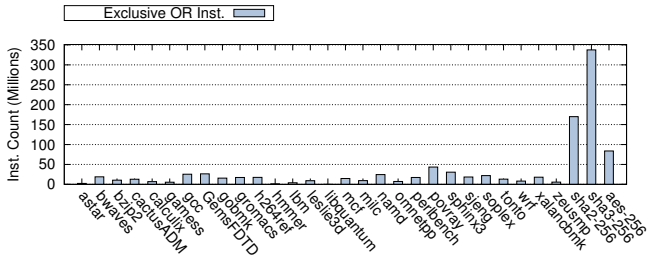


Fig. 7: Number of Exclusive OR (XOR) instructions (in millions).

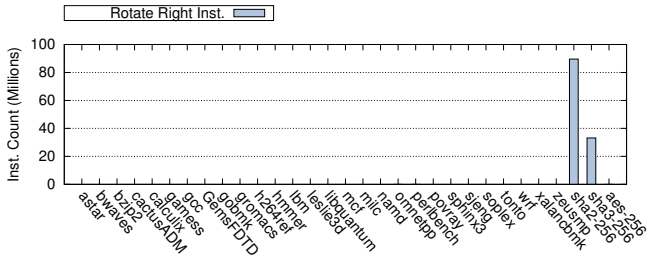


Fig. 8: Number of Rotate Right (RR) instructions (in millions).

in SHA-2. This is evident by the 28M instruction count we observe for SHA-2. However, despite SHA-2 having an SR count that is 10x higher than the benchmarks in the SPEC2K6, we find that AES exhibits 2.7x the amount of shift right operations compared to SHA-2. With the exception of SHA-3, we observe that cryptographic functions have significantly more SR instructions relative to the various SPEC2K6 benchmarks. However, this is not the case for the shift left (SL) instruction. For instance, Figure 6 shows that *libquantum* has 9x and 3x the number of SL instruction compared to SHA-3 and AES, respectively. As a result, SL instructions are an unsuitable feature on their own for cryptojacking detection.

Unlike the previous instructions, we find that hash functions exhibit a relatively high number of XOR instructions compared to benchmarks in the SPEC2K6 suite. We observe an XOR count of 170M and 337M for the SHA-2 and SHA-3 functions, respectively. This is 4x and 8x higher than *povray* which had the highest XOR count in the SPEC2K6 suite. Furthermore, despite the XOR operation being fundamental to each round of the AES encryption algorithm, we record 84M XOR instructions. Both SHA-2 and SHA-3 execute 2x and 4x the amount of XOR instructions relative to AES over the same billion instruction window. This is illustrated in Figure 7.

Figures 8 and 9 illustrate the count for the rotate right and left instructions (RR and RL). We find an RR count of 89M and 33M in the case of SHA-2 and SHA-3, respectively. However, unlike the aforementioned cryptographic functions, we observe 0 to 15 RR instructions across the rest of the workloads with *perlbench* having the highest count. AES, on the other hand, only had 3 RR instructions. This is illustrated in Figure 8. Similarly, we see low occurrences of the RL instruction across the different SPEC2K6 benchmarks in addition to AES. On average, we observe 85 RL instruction across the aforementioned workloads with perlbench having the highest count (1590 instructions). AES only had 37 RL instructions. Overall, we find that RR and RL instructions serve as excellent features for identifying cryptojacking behavior.

B. Mitigating Code Obfuscation Attacks

Our results demonstrate that rotate instructions in particular represent excellent features for distinguishing between benign and cryptojacking workloads. However, it is conceivable that an attacker would use code obfuscation techniques that combine different sequences of instructions to fulfill the same rotation functionality as a way of subverting detection. An attacker could replace n-bit left rotations (R_l^n) with a sequence of n-bit left shifts (S_l^n), or ($\vee$) operations, and m-bit right shifts (S_r^m), as illustrated by Equation (6a). A similar approach could be applied for replacing an n-bit right rotation as shown in Equation (6b).

$$R_l^n = S_l^n \vee S_r^{64-n} \tag{6a}$$

$$R_r^n = S_r^n \vee S_l^{64-n} \tag{6b}$$

Furthermore, an attacker could use a mix of rotate instructions and their shift/or equivalents in the same code, as well as a combination of the aforementioned instructions using different rotate/shift directions (left/right). Therefore, to mitigate code obfuscation attacks, we propose using a single performance counter. In other words, we only track the cumulative sum of all rotate, shift, and exclusive or instructions. Since our solution uses a programmable set of instructions that can be updated via microcode, the design can effectively track the appropriate instructions by tagging them at the instruction decode stage. This approach also has

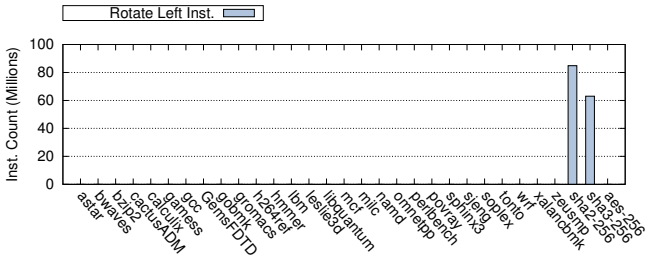


Fig. 9: Number of Rotate Left (RL) instructions (in millions) after executing 1 billion instructions.

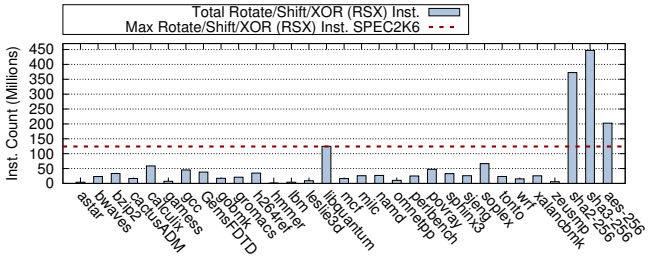


Fig. 10: Total number of Rotate (RR/RL), Shift (SR/SL), and Exclusive OR (XOR) instructions (in millions).

the added benefit of keeping the hardware simple by only requiring a single counter for the entire solution as opposed to dedicating a different performance counter for each instruction type.

Figure 10 shows the cumulative number of rotate left (`RL`), rotate right (`RR`), shift left (`SL`), shift right (`SR`), and exclusive or (`XOR`) after executing the benchmarks for 1 billion instructions. We refer to these as `RSX` instructions (rotate, shift, and exclusive or). We find that the secure hash algorithms, SHA-2 and SHA-3, exhibit an `RSX` count that is 3x and 3.5x higher relative to the `libquantum` benchmark. Despite `libquantum` being the benchmark with the highest total of `RSX` instructions in the SPEC2K6 suite, its count is well below what we record for SHA-2 and SHA-3. For illustration purposes, Figure 10 includes the cumulative count for AES separately. However, on a deployed system, the number of accumulated instructions in the AES workload (202M) would be combined with the count collected from the SHA-2/SHA-3 workloads. This is due to the fact that anonymous cryptocurrencies, including Monero, incorporate encryption in their algorithms as part of fulfilling their anonymity requirement.

In addition, an attacker may choose to encode a subset or all `XOR` instructions using the `OR` operator (e.g. $A\bar{B} \vee \bar{A}B$). In this scenario, our `RSX` approach could be adapted to encompass the `OR` instruction (`RSXO`). The cumulative count of `RSXO` instructions is illustrated in Figure 11. We observe that the `RSXO` count for SHA-2 and SHA-3 scales to 7x and 9x relative to the `libquantum` benchmark. This result underscores the ability of our solution to scale to different instructions while remaining effective in detecting cryptojacking activity.

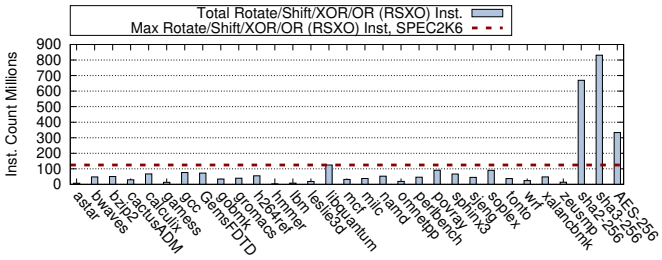


Fig. 11: Total number of Rotate (RR/RL), Shift (SR/SL), Exclusive OR (XOR), and OR instructions (in millions).

C. Characterization of Standard User Applications

To further validate our approach, we conducted experiments using various user applications that span multiple categories. We used a diverse set of applications from four different categories: social, communication, productivity, and entertainment. The different categories and their corresponding applications used in this experiment are summarized in Table II. Applications in each category were run interactively for a period of one hour. This activity included streaming live video content over a browser, creating and saving a variety of documents, reading emails, and playing games. The cumulative rotate, shift, and exclusive or instructions each application generated as a result of the interaction is summarized in Figure 15.

Hash Instructions in Standard User Applications. We observe that on average, user workloads have 0.3 billion `RSX` instructions with five applications standing out in count: Ramme, Angry Birds, Everpad, WhatsDesk, and Slack. Ramme is a social media application that is equivalent to the popular Instagram program. This application exhibits the highest count across all user workloads with its overall `RSX` count reaching 5.2B. A close examination of this application shows 77% of the `RSX` operations were shift related. On the other hand, 20% of the remaining instructions were exclusive or operations. Angry Birds, a game application, exhibits a different trend. The vast majority of the `RSX` instructions were exclusive or operations (61%), while only 33% of the instructions were shift operations. Everpad, a note taking app that is equivalent to the popular Evernote program, had a total of 2.1B `RSX` instructions. A closer look into the Everpad application reveals that over 69% of the `RSX` instructions were shift operations. On the other hand, 31% of the remaining instructions were exclusive or operations. We observe a similar trend with the WhatsDesk application, a desktop version of WhatsApp. We recorded a slightly reduced total of 1.3B `RSX` instructions. We observe a similar trend in terms of the distribution of the `RSX` instructions. We find that during execution, WhatsDesk exhibited 67% and 31% shift and exclusive or operations, respectively. The Slack application, another messaging platform, observed a slightly lower `RSX` count relative to WhatsDesk. We find that during execution, that application had a total of 0.9B `RSX` instructions. Although Slack had a lower `RSX` count, it had a higher number of shift operations. We observed that over 86% of the instructions were shift operations while 14%

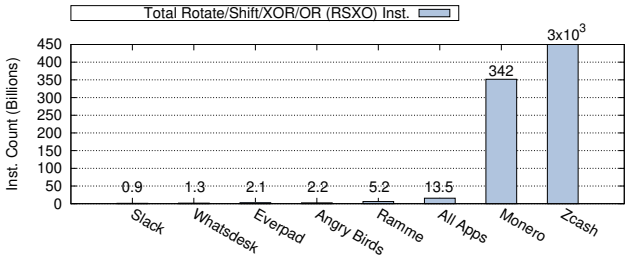


Fig. 12: RSX instructions in Monero, Zcash, and user applications in Table II after a one hour execution period.

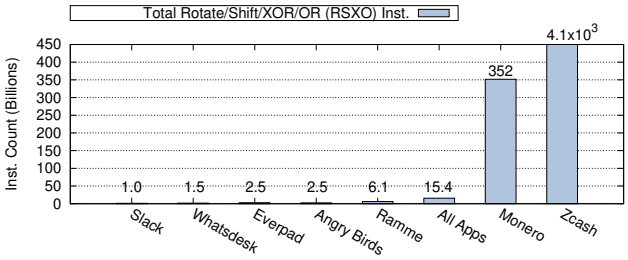


Fig. 13: RSXO instructions in Monero, Zcash, and user applications in Table II after a one hour execution period.

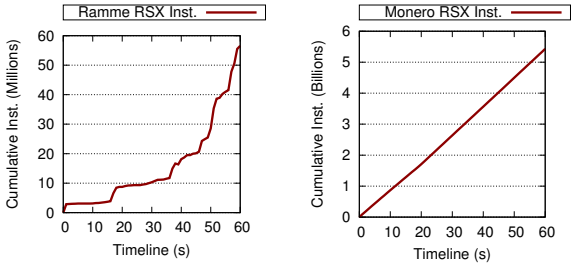
of the instructions were exclusive or operations.

Standard Applications vs. Crypto Mining Services. We compared execution traces from Monero and Zcash mining services to the user applications listed in Table II. All applications were executed for one hour. Figure 12 summarizes these results. Overall, we observe that cryptocurrency mining workloads had a significantly higher RSX count relative to the remaining user applications. For instance, we observe that Monero had 342B RSX instructions over its one hour execution period. This is more than 65x the RSX count relative to Ramme which exhibited the highest RSX count among the user applications shown in Figure 15, and more than 155x the RSX count relative to Angry Birds. We observe an even higher RSX count in the case of Zcash. For example, Zcash shows an RSX count that is three orders of magnitude higher than Ramme while executing over the same one hour period. Even when combining the RSX instructions of all the user applications shown in Figure 15 which amounts to less than 14B, the count is still 26x and 230x smaller than the count observed by Monero and Zcash, respectively. While our proof-of-concept focuses on evaluating the solution against the core hash instructions (RSX), we show that a similar trend can be observed when adapting our design to track RSXO instructions. Figure 13 shows that even when combining the RSXO instructions of all the user applications shown in Figure 13, Monero and Zcash's RSXO count is 23x and 265x higher.

To better understand the contribution of the core hash instructions (RSX) to user applications and cryptocurrency services, Table III shows a breakdown of such instructions over a one hour execution period. In addition to Monero and Zcash, Table III shows the top five user applications with the most RSX instructions, as well as, the combined cumulative count for the remaining applications listed in Table II. We observe that over an execution period of one hour, rotate instructions are primarily present in Monero and Zcash. All other applications show less than one billion rotate instructions. In the case of shift instructions, we find that with the exception of Monero and Zcash, only Ramme and Everpad show an instruction count that is above one billion. For instance, even when we examine the overall count of shift instructions from all the remaining user applications accumulated together, we observe 0.6B instructions. We see a similar trend for exclusive or instructions. We find that none of the applications' XOR

Application	Rotate	Shift	XOR	Total RSX
Monero	83.1	10.2	248.3	341.6
Zcash	27.9	$1.2 \cdot 10^3$	$1.8 \cdot 10^3$	$3.0 \cdot 10^3$
Slack	0.0	0.8	0.1	0.9
WhatsDesk	0.0	0.9	0.4	1.3
Everpad	0.0	1.5	0.7	2.1
Angry Birds	0.2	0.7	1.3	2.2
Ramme	0.1	4.1	1.1	5.2
Remaining	0.0	0.6	0.7	1.3

TABLE III: Summary of RSX instruction breakdown in billions across user and cryptocurrency applications.



((a)) Ramme ((b)) Monero

Fig. 14: Total RSX count over a one execution minute period for (a) Ramme and (b) Monero.

instructions exceed the one billion mark except for Ramme and Angry Birds. Finally, a close examination of the RSX operations for Monero and Zcash show that the vast majority of the instructions are in the form of XOR operations. For instance, the XOR operation represents 73% of Monero's RSX instructions. A similar trend is observed with Zcash where more than 59% of the RSX count are XOR instructions. We attribute the elevated count in XOR instructions to the encryption portion of the anonymous cryptocurrency algorithms. Overall, the aforementioned data illustrates the effectiveness of using RSX instructions as features for cryptojacking detection.

Figure 14 shows the cumulative count for rotate, shift, and exclusive or instructions over an execution period of one minute. We observe that the RSX for Monero is significantly higher than what we record for Ramme. However, in order to determine a suitable threshold value that results in a low false positive rate, we tested a total of 153 user applications and benchmarks with different threshold values over a one minute

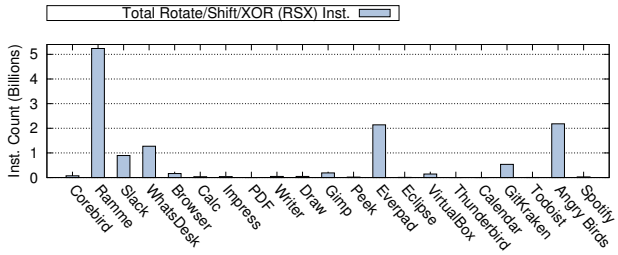


Fig. 15: RSX instructions in real user applications after a one hour execution period.

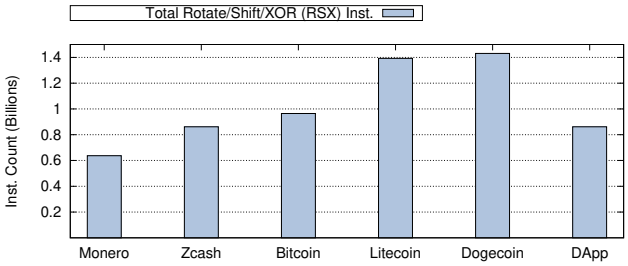


Fig. 16: RSX instructions in non-mining crpytocurrency applications after a one hour execution period.

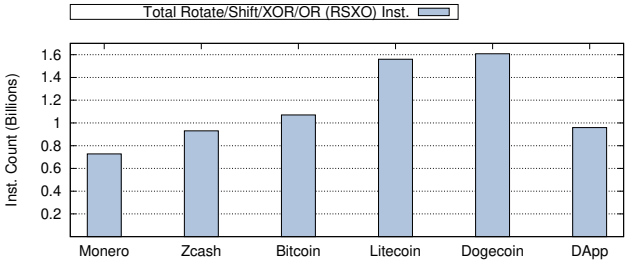


Fig. 17: RSXO instructions in non-mining crpytocurrency applications after a one hour execution period.

execution period. Based on these experiments, we found that selecting an RSX threshold of 2.5B inst./min enables us to detect cryptojacking behavior involving the Monero and Zcash cryptocurrencies with an accuracy of 100%. Furthermore, we observed that false positives occur only when the core cryptographic functions (AES, SHA-3, and SHA-2) themselves are run uninterrupted for extended periods. Even under such circumstances, the false positive rate remains below 2%. In general, running cryptographic functions for an extended period is atypical and is often indicative of malicious activity. For instance, continuously running AES on a system may be indicative of ongoing ransomware activity. Nonetheless, this can still legitimately occur in some infrequent cases, such as a user encrypting a large file for confidentiality purposes. As a result, we consider these cryptographic functions as part of our evaluation for false positives.

D. Non-mining Cryptocurrency Applications

In addition to standard applications that are commonly used by consumers, we tested a different type of user applications that are cryptocurrency focused. We evaluated non-mining cryptocurrency applications including different crypto-wallets that were configured to issue transactions to live services over a period of one hour. In addition to crypto-wallets, we tested a decentralized application implemented in Solidity that interacted with a smart contract over a period of one hour. Figure 16 shows the results of this experiment. We observe that the RSX count across the different crypto-wallet applications (shown with -W appended to each cryptocurrency type) ranges between 0.6B and 1.4B. This is 4.1x to 9.7x less than the social media application, Ramme. Similarly, we observe an RSX count of 0.9B for the decentralized application shown as DApp in Figure 16. We see a similar trend for the aforementioned applications when considering the RSXO count. In this case, the RSXO ranges between 0.7B and 1.6B, which is significantly less than what we recorded for the application Ramme. This is illustrated in Figure 17.

E. Impact of Throttling on Detection

Prior work has shown that cryptojacking campaigns rely on throttled execution as a way of evading detection [5], [17]. When attackers use throttling, they often set the throttle rate to 30%. In other words, only 70% of the system's processing power is used [5]. To this end, we evaluate our solution's susceptibility to evasion techniques that leverage throttled execution. In our evaluation, we periodically track the count of RSX instructions for each process against a 2.5B RSX inst./min threshold as part of classifying workloads as either malicious or benign. To put things in perspective, Monero has an RSX rate of 5.7B instructions per minute. Therefore, our solution is able to detect cryptocurrency mining activity when the most common throttling rate of 30% is used. Furthermore, our solution can detect such activity with throttling rates that exceed 50%.

In addition to RSX instructions, we explored supplemental features that could be added in order to make our design more resilient to throttling attacks. To this end, we considered multiple machine learning algorithms with a dataset that consisted of 272 samples. We used the Principle Component Analysis (PCA) algorithm to reduce the initial feature size from 527 to 11. The reduced feature set included load instructions such as MOV, MOVSS, and MOVSD, as well as, arithmetic instructions such as IMUL and ADD. Figure 18 summarizes the detection rate of the tested models as a function of throttling rates. All models performed well for throttling rates that ranged between 10% – 50%. However, we observed that SVM yielded the best performance, providing a detection rate of 100% even when execution was throttled by 95%. Logistic Regression also offered similar detection capability for the 95% throttling rate. However, this algorithm had a very high false positive rate (FPR) of 40%. SVM, on the other hand, maintained an FPR that was under 2% when execution was throttled by 95%. Despite the robustness of our solution, an attacker may use aggressively low throttling rates that are beyond 95% in order

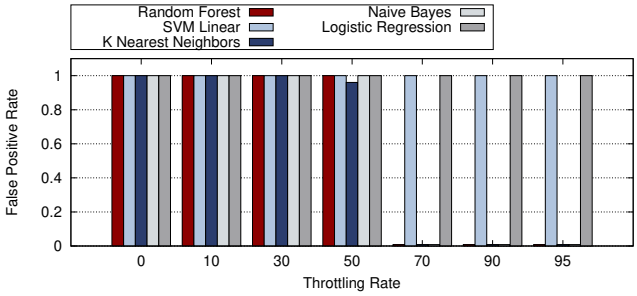


Fig. 18: Detection rate as a function of throttling rates when combining machine learning models with more features.

CPU Utilization	*Profit/Hour (XMR)*	*Profit/Hour (USD)*
100%	0.142	32.78
75%	0.106	24.58
50%	0.071	16.39
25%	0.035	8.194
5%	0.007	1.639
1%	0.001	0.328

TABLE IV: Estimated profit for different throttling rates.

to evade our solution. Although this is possible, we show that the profitability of this approach diminishes significantly. Table IV illustrates estimated profits for different throttling rates. For instance, if an attacker uses throttling rates between 1% – 5%, the profit per hour is estimated to be between $0.3 – $1.6. We believe such returns on investment would render cryptojacking unprofitable for attackers.

F. Performance Overhead

Overall, our solution is lightweight and incurs insignificant overhead. We observe that all SPEC2K6 workloads exhibit less than 1% overhead. The benchmark that exhibits the largest overhead is *omnetpp* which experienced a 0.7% reduction in performance. We observe a similar trend with *povray* which experienced a 0.6% reduction in performance. All of the remaining workloads had overheads that are below 0.5%. Such low overheads underscore the efficiency of our approach.

VII. Related Work

Multiple bodies of work [5], [14], [17], [20], [21], [38] explored different solutions in an effort to mitigate cryptojacking attacks. Work by Hong et al. [17] explored a behavior-based approach that relies on two runtime profilers for detecting cryptojacking activity. In their work, the first profiler uses a signature-based approach for detecting incoming cryprojacking scripts. Their second phase, on the other hand, makes use of a profiler that analyzes the call-stack of mining scripts and looks for hotspot calling contexts in order to detect cryptojacking activity. They achieve this by tracking calls into commonly used hash libraries that are offered by browsers. Kharraz et al. [20] analyzed Alexa Top 1M websites in order to construct a dataset of benign and malicious JavaScript code. They then applied machine learning algorithms, such as SVM and random forest for classifying JavaScript prior to its execution. They showed that their mechanism can achieve a 97% detection rate. Unlike our work, these solutions are limited to detecting cryptojacking activity that is confined to the web browser.

Work by Konoth et al. [21] proposed the use of WebAssembly (Wasm) to construct and detect cryptographic hashing activity within a browser. They achieved this by performing static analysis of Wasm. Unfortunately, this approach introduces overhead to the browser's execution model since it requires static analysis [38]. It also makes the solution limited to detecting cryptojacking code that is implemented in Wasm. Furthermore, the solution makes use of performance counters within the last level cache (LLC). Since the LLC is a common resource across multi-core processors that could be running other programs simultaneously, the values of the counters could be significantly skewed. Similarly, work by Tahir et al. [35] explored a performance counter-based approach. However, this work makes use of generic performance counters such as page faults, and LLC access counters that are not tuned to cryptographic workloads, which can lead to false positives. Lachtar et al. [22] explored the initial idea of using a hardware-based detector that our work builds upon.

Additional work by Eskandari et al. [13] examined Monero mining through Coinhive. They found that a domain parking service represented one of the biggest Coinhive campaigns that ran Coinhive on more than 11,000 parked websites. Mining Hunter [30], focused on the development of a crawler that analyzed Javascript code in order to detect mining scripts. Their framework was able to find several cryptojacking campaigns including one that infected over 1100 websites through malicious advertisement scripts. Liu et al. [23] developed a solution that examined compiled JavaScript code within the Chrome browser, while RAPID by Rodriguez et al. [31] explored a learning-based approach for detecting in-browser cryptojacking. They compared six different approaches that relied on API calls and showed that their mechanism could achieve a precision of 96% after analyzing Alexa top 330,550 sites. Virtually, all prior work that we are aware of focused on detecting cryptojacking activity within the browser. Our work, explores a generic end-to-end approach that enables detection in an application agnostic fashion.

VIII. Conclusion

This paper presents an application agnostic design that harnesses innovations at the microarchitecture and OS layers for defending against cryptojacking attacks. We demonstrate that a select set of instructions that are commonly employed in cryptographic functions is sufficient for fingerprinting cryptojacking activity. Our solution incurs minimal performance impact and is resilient to multiple evasion techniques.

Acknowledgements

The authors would like to thank our shepherd Fengwei Zhang and the anonymous reviewers for their feedback and comments on this work. This work was funded in part by the National Science Foundation under grant CNS-1947580.

REFERENCES

[1] Advanced Encryption Standard. *National Institute of Standards and Technology, Federal Information Processing Standards Publication 197*, November 2011.

[2] Cryptojacking surges in popularity growing by 31% over the past month. https://adguard.com/en/blog/november_mining_stats.html.

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

[4] The Avast Report. https://blog.avast.com/greedy-cybercriminals-host-malware-on-github/.

[5] Hugo LJ Bijmans, Tim M Booij, and Christian Doerr. Inadvertently making cyber criminals rich: a comprehensive study of cryptojacking campaigns at internet scale. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1627–1644, 2019.

[6] Hugo LJ Bijmans, Tim M Booij, and Christian Doerr. Just the tip of the iceberg: Internet-scale exploitation of routers for cryptojacking. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 449–464, 2019.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[8] A Biryukov and D Khovratovich. Equihash (2016). asymmetric proof-of-work based on the generalized birthday problem. NDSS.

[9] Andrew Brandt. The persistent nuisance of cryptomining looks set to grow. https://news.sophos.com/en-us/2018/09/21/the-persistent-nuisance-of-cryptomining-looks-set-to-grow.

[10] Catalin Cimpanu. A crypto-mining botnet has been hijacking MSSQL servers for almost two years. https://www.zdnet.com/article/a-crypto-mining-botnet-has-been-hijacking-mssql-servers-for-almost-two-years, April 1 2020.

[11] Catalin Cimpanu. Supercomputers hacked across Europe to mine cryptocurrency. https://www.zdnet.com/article/supercomputers-hacked-across-europe-to-mine-cryptocurrency, May 16 2020.

[12] CryptoNight . https://cryptonote.org.

[13] Shayan Eskandari, Andreas Leoutsarakos, Troy Mursch, and Jeremy Clark. A first look at browser-based cryptojacking. In *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 58–66. IEEE, 2018.

[14] Fábio Gomes and Miguel Correia. Cryptojacking detection with cpu usage metrics. In *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*, pages 1–10. IEEE, 2020.

[15] Nils Grossberg. Turkey launching its national cryptocurrency. https://dagcoin.org/turkey-launching-its-national-cryptocurrency.

[16] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[17] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Haixin 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*, pages 1701–1713, 2018.

[18] Ionut Ilascu. Cryptojacking android apps continue to plague google play store. https://www.bleepingcomputer.com/news/security/cryptojacking-android-apps-continue-to-plague-google-play-store.

[19] Ulya R Karpuzcu, Brian Greskamp, and Josep Torrellas. The bubblewrap many-core: popping cores for sequential acceleration. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 447–458, 2009.

[20] Amin Kharraz, Zane Ma, Paul Murley, Charles Lever, Joshua Mason, Andrew Miller, Nikita Borisov, Manos Antonakakis, and Michael Bailey. Outguard: Detecting in-browser covert cryptocurrency mining in the wild. In *The World Wide Web Conference*, pages 840–852, 2019.

[21] Radhesh Krishnan Konoth, Emanuele Vineti, Veelasha Moonsamy, Martina Lindorfer, Christopher Kruegel, Herbert Bos, and Giovanni Vigna. Minesweeper: An in-depth look into drive-by cryptocurrency mining and its defense. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1714–1730. ACM, 2018.

[22] Nada Lachtar, Abdulrahman Abu Elkhail, Anys Bacha, and Hafiz Malik. A cross-stack approach towards defending against cryptojacking. *IEEE Computer Architecture Letters*, 19(2):126–129, 2020.

[23] Jingqiang Liu, Zihao Zhao, Xiang Cui, Zhi Wang, and Qixu Liu. A novel approach for detecting browser-based silent miner. In *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, pages 490–497. IEEE, 2018.

[24] Malwarebytes. Cryptojacking. https://www.malwarebytes.com/cryptojacking, June 19 2020.

[25] The Monero Project. https://web.getmonero.org/the-monero-project.

[26] Charlie Osborne. Smominru hijacks half a million PCs to mine cryptocurrency, steals access data for Dark Web sale. https://www.zdnet.com/article/new-cryptojacking-campaign-strikes-half-a-million-pcs, August 7 2019.

[27] Danny Palmer. Android security: Cryptocurrency mining-malware hidden in VPNs, games, and streaming apps, dowloaded 100,000 times. https://www.zdnet.com/article/android-security-cryptocurrency-mining-malware-hidden-in-vpns-games-and-streaming-apps-dowloaded, April 5 2018.

[28] Danny Palmer. Cybercriminals spotted hiding cryptocurrency mining malware in forked projects on GitHub. https://www.zdnet.com/article/cybercriminals-spotted-hiding-cryptocurrency-mining-malware-in-forked-projects-on-github, March 15 2018.

[29] Kari Paul. Thanks to hackers, you might be mining cryptocurrency without realizing it. https://www.marketwatch.com/story/how-hackers-can-turn-your-phone-into-a-cryptomine-2018-05-11.

[30] Julian Rauchberger, Sebastian Schrittwieser, Tobias Dam, Robert Luh, Damjan Buhov, Gerhard Pötzelsberger, and Hyoungshick Kim. The other side of the coin: A framework for detecting and analyzing web-based cryptocurrency mining campaigns. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–10, 2018.

[31] Juan D Parra Rodriguez and Joachim Posegga. Rapid: Resource and api-based detection against in-browser miners. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 313–326, 2018.

[32] Markku Juhani Saarinen and Jean-Philippe Aumasson. The blake2 cryptographic hash and message authentication code (mac). *Internet Engineering Task Force*, 2015.

[33] Yuriy Shiyanovskii, F Wolff, Aravind Rajendran, C Papachristou, D Weyer, and W Clay. Process reliability based trojans through nbti and hci effects. In *2010 NASA/ESA Conference on Adaptive Hardware and Systems*, pages 215–222. IEEE, 2010.

[34] Statista. Average number of daily cryptocurrency transactions in 3rd quarter of 2019, by type. https://www.statista.com/statistics/730838/number-of-daily-cryptocurrency-transactions-by-type.

[35] Rashid Tahir, Muhammad Huzaifa, Anupam Das, Mohammad Ahmad, Carl Gunter, Fareed Zaffar, Matthew Caesar, and Nikita Borisov. Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 287–310. Springer, 2017.

[36] Ady Tal. Intel software development emulator, 2020.

[37] Abhishek Tiwari and Josep Torrellas. Facelift: Hiding and slowing down aging in multicores. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 129–140. IEEE, 2008.

[38] Wenhao Wang, Benjamin Ferrell, Xiaoyang Xu, Kevin W Hamlen, and Shuang Hao. Seismic: Secure in-lined script monitors for interrupting cryptojacks. In *European Symposium on Research in Computer Security*, pages 122–142. Springer, 2018.

[39] Tommy Dong Yuanjing Guo. Several Cryptojacking Apps Found on Microsoft Store. https://symantec-blogs.broadcom.com/blogs/threat-intelligence/cryptojacking-apps-microsoft-store.

[40] Zcash: Privacy-protecting digital currency, 2020.