CryptojackingTrap: An Evasion Resilient Nature-Inspired Algorithm to Detect Cryptojacking Malware

Atefeh Zareh Chahoki, Hamid Reza Shahriari, Marco Roveri

***Abstract*—The high profitability of mining cryptocurrencies mining, a computationally intensive activity, forms a fertile ecosystem that is enticing not only legitimate investors but also cyber attackers who invest their illicit computational resources in this area. Cryptojacking refers to the surreptitious exploitation of a victim's computing resources to mine cryptocurrencies on behalf of the cyber-criminal. This malicious behavior is observed in executable files and browser executable codes, including JavaScript and Assembly modules, downloaded from websites to victims' machines and executed. Although there are numerous botnet detection techniques to stop this malicious activity, attackers can circumvent these protections using a variety of techniques. In this paper, CryptojackingTrap is presented as a novel cryptojacking detection solution designed to resist most malware defense methods. The CryptojackingTrap is armed with a debugger and extensible cryptocurrency listeners and its algorithm is based on the execution of cryptocurrency hash functions: an indispensable behavior of all cryptojacking executors. This algorithm becomes aware of this specific hash execution by correlating the memory access traces of suspicious executables with publicly available cryptocurrency P2P network data. With the advantage of this assembly-level investigation and a nature-inspired approach to triggering the detection alarm, CryptojackingTrap provides an accurate, evasion-proof technique for detecting cryptojacking. After experimental evaluation, the false negative and false positive rates are zero, and in addition, the false positive rate is mathematically calculated as 10^{-20}. CryptojackingTrap has an open, extensible architecture and is available to the open-source community.**

***Index Terms*—Blockchain, cryptocurrency, Bitcoin, Ethereum, Monero, mining, malware detection, botnet, Botcoin, Cryptojacking, blockchain security, dynamic analysis, dynamic binary instrumentation.**

I. Introduction

FLOURISHING ecosystems of cryptocurrencies, captivating investors' attention, present an irresistible opportunity for cyber-criminals to monetize their resources. This includes leveraging the widespread bot infrastructure and exploiting victims' resources in mining [1]. The focus of this work is *cryptojacking*, a term used to describe the surreptitious utilization of a user's computing resources to mine cryptocurrencies on behalf of the cyber-criminal, also known as *Coinjacking*, *Cryptomining* or *drive-by mining*. Its descendent type that exclusively mines Bitcoin cryptocurrency was named *botcoin* [2]. The consequences of cryptojacking attacks would be financially significant for both individuals and businesses as it would result in system slow-down, the decreased lifespan of hardware, and increased electricity expenses. Botnets have a variety of behavioral features in higher-level classifications of propagation, rallying mechanism, command, and control (C&C) communication, purpose, evasion, and topology [3]. Cyber-security professionals enhance security products based on the botnet features in different life cycles to trap and protect the final user devices from exploitation. These features are extracted from significant patterns observed in the under-attack network or host. Although extensive literature exists for detecting botnets, the attack and counter-attack mechanisms associated with them are continuously evolving due to the emergence of new technologies.

In June 2011, the first Bitcoin mining malicious software was uncovered in the wild [4]; Afterward, countless malware families have adopted cryptocurrency mining. The first botcoin detection reports used general botnet feature detection methods. Although all these methods benefit from vast experience in botnet detection, they did not use the unique features of miners to increase their detection precision. In [5], authors introduced BotcoinTrap, which exploits the memory access trace partial predictability feature of botcoins. Unlike other approaches such as MineSwipper [6], BotcoinTrap can detect botcoins in executable files or running processes, and cannot be bypassed by changing general botnet features, including C&C protocols features. However, BotcoinTrap is limited to detecting Bitcoin mining malware and lacks precision metrics calculations for the proposed algorithm. In this paper, these limitations are addressed, and implementation is significantly extended with numerous evaluations to introduce a new approach called CryptojackingTrap, capable of detecting cryptojacking activity on any cryptocurrency network.

In this work, CryptojackingTrap is proposed, drawing inspiration from the Venus Flytrap (Dionaea muscipula) plant that has evolved a unique mechanism to capture insects. Detection of prey movement is achieved through tiny trigger hairs on its leaves, which, when stimulated, initiate a rapid trap closure. Intriguingly, the Venus Flytrap has a 20-second time window for a second strike, which has been optimized to maximize capture rates while minimizing energy expenditure [7]. As computer scientists, inspiration is derived from this efficient mechanism to develop a novel approach for detecting cryptojacking malware. The proposed method analyzes to identify

Manuscript received December 25, 2023; revised ????. *(Corresponding author Atefeh Zareh Chahoki).*

A. Zareh and M. Roveri are with *Dept. of Engineering and Computer Science* of the *University of Trento*, Trento, Italy (e-mail: atefeh.zareh@unitn.it, marco.roveri@unitn.it).

H. R. Shahriari is with the *Amirkabir University of Technology Tehran*, Tehran, Iran (e-mail: shahriari@aut.ac.ir).

malicious behavior triggered by suspicious events within an optimized window, intending to reduce both false positive and false negative rates, akin to the Venus Flytrap's optimization for survival. This research goal is to find cryptojacking detection techniques that cannot be bypassed using known evasion approaches (II) and his paper makes the following contributions: (1) A novel approach and efficient algorithm named *CryptojackingTrap* is introduced for detecting cryptojacking malicious behavior in suspicious executables, including executable files and processes. This approach is resilient against malware evasion techniques including obfuscation, encryption, and hash rate reduction. Its design is open-to-extend, and it supports further cryptocurrencies without requiring changes to core modules, due to the incorporation of lightweight plugins. (2) The entire solution has been implemented, comprising 6.5K SLOC (source lines of code), and the source code is made available to the research and industrial communities under an open-source license. (3) A mathematical analysis has been conducted to calculate the false positive rate for CryptojackingTrap detection, yielding a value of 10^{-20} for a typical application. (4) Empirical evaluation results confirm the precision of this approach across five dataset categories, totaling over 15 GB of public data. (5) A marked field is proposed for other cryptocurrency block data structures. This marked data can be used in blockchains where significant predictable data is lacking, enabling the CryptojackingTrap solution to still function effectively. The marked field facilitates the CryptojackingTrap with seed data to track memory access and ensure successful detection. The remainder of this paper is organized as follows: The problem statement is presented in Section II, Section III provides the most relevant background concepts used throughout the paper. Section IV is dedicated to elaborating this novel approach. Section V evaluates this approach mathematically and experimentally. Comparing this research contribution with other researchers' related work is done in Section VI and conclusions are discussed in Section VII.

II. Problem Statement

To determine the scale of the problem, researchers have analyzed CPU usage and campaign size. For instance, Zimba et al. [8] estimated that 32 percent of U.S. users are vulnerable to browser-based crypto mining. Hong et al. [9] found that approximately 10 million web users are affected by cryptojacking every month, resulting in a daily cost of 59,000 USD due to an additional 278K kWh of power consumption. Furthermore, [6] estimated that the profit of each cryptojacking campaign ranges from 14.36 USD to 31,060.80 USD per month on average, while [10] estimated an average profit of 340 USD per campaign per day (about 10,200 USD per month).

The malware detection problem becomes more complicated when attackers use techniques to evade detection tools. Botnet detection mechanisms and, consequently, evasion mechanisms target different life-cycle stages of botnets. Life-cycle stages are botnet conception, recruitment, interaction, marketing, attack execution, and attack success [11]. (1) A dynamic analysis of network traffic can be used to analyze the interactions between a botmaster and slaves by extracting features and behaviors on the network. However, the connection and interaction stages may be susceptible to bypassing if the C&C communication is directed through an encrypted channel. (2) Static analysis approaches over malware code are vulnerable to code obfuscation. Dynamic analysis of OS APIs for calling cryptography functions including hash is not effective if malware developers do not use OS APIs and develop their own customized code for calculations. (3) Static analysis of cryptography functions by focusing on finding cryptography function constants is another detection approach but it is not resilient to code obfuscation[12]. (4) Since cryptojacking is the most process-demanding attack, CPU activity analysis is also targeted as a detection solution. This security solution can still be evaded by decreasing the mining rate and dedicating that portion of botnet resource capacity to less computation-demanding tasks such as spam, click fraud, DDoS, identity theft, and information phishing. To maximize profits, botnet herders run resource-disjoint monetization activities simultaneously [13]. (5) Static code analysis to find constants related to mining pools keywords, and URLs is an alternative approach for detecting cryptojacking attacks. However, this approach is not resilient to code obfuscations.

III. Background

The glory of clear consensus over the world's most critical asset, money, without trusting any trusted party is rooted in blockchain and consensus protocols. Blockchain is a ledger of timestamped blocks of transactions chained by persisting the previous block hash in each block and the network is maintained by miners. They do a computation-intensive mathematical activity called mining which demands massive hash calculations on the block. In this section, detailed information on block structure is investigated in Bitcoin, Ethereum, and Monero. The targeted goal is to find predictable consecutive fields in the next valid block that is not broadcast on the network yet. These fields are clues to detect executables that deal with them very frequently in processing and memory retrievals. Amongst all cryptocurrencies' block parts, solely block header is potentially partially predictable which is elaborated in Table I. Other parts due to their diversity based on miners' transactions selection are not the subject of this section elaboration.

IV. CryptojackingTrap

This section elaborates CryptojackingTrap, an evasion resilient approach to detecting cryptojacking mining across various cryptocurrencies. Cryptojacking botnet matches general botnet life cycle stages as described in [11]. After examining different detection approaches, it was concluded that the life-cycle stage with the least diversity in evading techniques for cryptojacking detection is the last one, where the attacker must utilize the victim's machine to perform hash function calculations. In this stage, irrespective of variations in other life-cycle stages, all miners exhibit the same behavior of mining on the victim's machine, which is inevitable.

TABLE I
BITCOIN (TOP), ETHEREUM (MIDDLE), AND MONERO (BOTTOM) BLOCK HEADERS

	Field	Purpose	E.U.T. for the miners	Size (Byte)
Bitcoin	**Version**	indicates the set of block validation rules that must be followed	for soft-fork and update	4
	Previous Block Header Hash	chains valid blocks and ensures approved blocks integrity	almost once a ten minutes per newly broadcast mined block	32
	Merkle Root Hash	derives from the current block transactions hashes	by chaining any transactions	32
	Time	counts the current timestamp in second after 1970-01-01T00:00 UTC	every second	4
	Bits	indicates the current proof of work hardship	calculated every ten minutes	4
	Nonce	finds a random number in a brute-force search for PoW	random and unpredictable	4
Ethereum	**Parent Hash**	chains valid blocks and ensures approved blocks integrity	almost once a twelve seconds	32
	Ommers Hash	specifies ommers block	by changing ommers	32
	Beneficiary	declares beneficiary address, i.e mining fee recipient	by changing any beneficiaries	20
	State Root	specifies head node of the state trie	by changing an account state	32
	Transaction Root	keeps track of transactions in the current block	by changing any transactions	32
	Receipts Root	declares the hash of the root node of recipients in the block transactions	by changing any recipients	32
	Logs Bloom	logs searchable information about the block transactions	by changing any transactions	256
	Difficulty	indicates the number of leading zeros in the current block's hash	dynamically is calculated	32
	Number	counts the ancestor blocks for the current block	by increasing blockchain length	32
	Gas Limit	limits the total gas that may be used by the current block	average 15M but miner defines	32
	Gas Used	declares the total gas that was used by the current block	by changing transactions	32
	Timestamp	equals the block creation time and is the output of Unix time()	every second	32
	Extra Data	adds extra data relevant for this block	arbitrary by miners	32
	Mix Hash	specifies the current block hash and must meet the difficulty constraints	by changing the block	32
	Nonce	finds a random number in a brute-force search for PoS	random and unpredictable	8
Monero	**Major Version**	specifies the major block header version (always 1)	constant value (1)	1
	Minor Version	serves as a voting mechanism	updated based on miner vote	1
	Timestamp	indicates block creation time (UNIX timestamp)	each second	8
	Prev Id	specifies the identifier of the previous block	on new mined block broadcast	32
	Nonce	finds a random number in a brute-force search for PoW	random and unpredictable	4

As expounded in Section II, network, code, and CPU analysis are subject to potential evasion techniques from encryption, obfuscation, and mining rate reduction, respectively. Regardless of the characteristics of the code, network traffic, or hash calculation rate, attackers must still retrieve data and calculate the hash. This behavior is essential and therefore forms the basis for trapping cryptojacking attacks.

Dynamic analysis is utilized to trace the memory retrieval contents that malware uses as input parameters for hash function calculation. This information is practically impossible to conceal and can be partially predicted at any time. The input for the hash function consists of the data structure of the cryptocurrency block, and it is not entirely predictable what this data will be on the victim's machine during an attack. This is because the block, including collected transactions, comprises flexible values that the cryptojacking miner determines. Nonetheless, the subsequent section explores which elements of a block can be predicted, and the formal evaluation presented in Section V-A demonstrates that this prediction is significant enough to serve as detection criteria.

A. Forecasting Attacks Data in Use

According to the E.U.T field on Table I, it can be concluded that the "previous block hash" possesses outstanding features. First, this field is shared in all cryptocurrency block headers, as the most fundamental chaining feature of blockchains is implemented using this chaining field. Second, the value of this field is predictable for the miner's block under development at any given time, as it must be equal to the hash of the last accepted block on the network that is publicly available. Lastly, the size of this field (32 bytes) is sufficient for correlating suspicious executable memory traces and network data. While some fields, such as the "major version" in Monero, are constant and predictable, they are not candidates due to their small size (1 byte).

The "version" field in Bitcoin is followed immediately by the "previous block header," resulting in a predictable 32+4 byte data sequence. In 2018, BotcoinTrap detected predictable data by assuming the block version as predictable, as the average active block versions per day were only two. However, this field is no longer predictable. As shown in Figure 1, Bitcoin blocks were broadcast with 124 different versions in one month (from 2/13/2020 to 3/13/2020). However, the diversity of versions in 2023 has increased significantly and is too vast to be represented in a pictorial diagram.

In consideration of the adoption of "previous block header hash" as the shared big predictable field for its algorithm, CryptojackingTrap is analyzed in Section V-A to possess

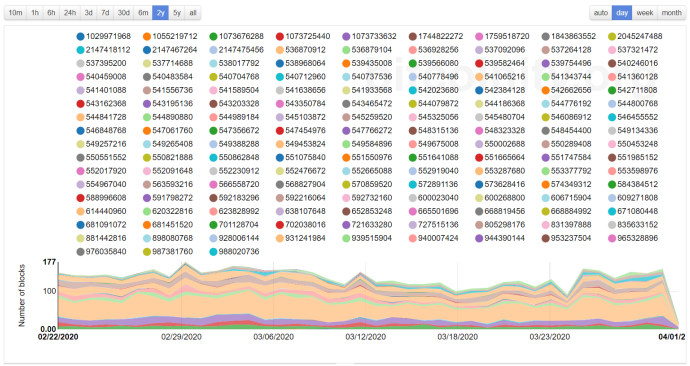

Fig. 1. Diverse Bitcoin block versions actively used by network [14]

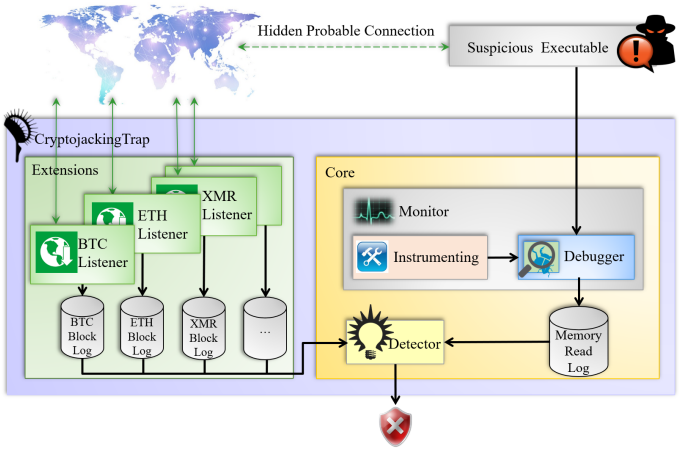

Fig. 2. Asynchronous architecture of CryptojackingTrap.

sufficient precision in detection. It is crucial to note that the term 'hash' used throughout the rest of this paper to describe a predictable field can be substituted with any other predictable field and calculations are not affected.

To tackle the challenge of multiple previous block hash in cryptocurrencies, particularly when dealing with orphan blocks, the authors proposed the incorporation of a dedicated "marked field" within the block data structures. This approach presents a streamlined and efficient solution that eliminates the necessity for cryptocurrency listeners and results in reduced computational overhead, as detection relies solely on the marked field. Importantly, this implementation seamlessly integrates with existing cryptocurrency implementations, requiring no significant change, and can be smoothly deployed for improved detection precision and performance. Another noteworthy suggestion is to incorporate this detection approach into hash function design, ensuring that the predictable field remains intact during hash method calculations and avoids the fragmentation that would otherwise necessitate fetching large segments of the predictable field from memory, thereby serving as a robust indicator for detecting miner malware.

B. The Architecture of CryptojackingTrap

In this section, the architecture of CryptojackingTrap is proposed in Fig. 2, comprising two primary blocks. The right block is labelled as "Core" which includes the main modules of CryptojackingTrap (monitor and detector), and the left as "Extensions" including listeners. In the following, different modules of the architecture are described:

Listener. The listeners are lightweight extensions, each one responsible for listening and gathering information from its specific blockchain P2P network. Each listener listens to its specific network block announcements and extracts valuable data, which is the previous block hash (Section IV-A), and stores it on the "Block Log (BL)" file(s). Block logs will be used for the rest of the detection process in the detector module. Since block logs have a predefined internal format that is also expected in the detector, developers can develop new listeners for the CryptojackingTrap solution without any need for chaining the core modules.

Monitor. On the right side of Fig. 2, the debugger module is dedicated to tracing the suspicious executable's activities. The type of activity that CryptojackingTrap is interested in monitoring, which is a memory read access, is defined in the instrumenting part of the monitor module. Consequently, the output of this module is the "Read Log (RL)" file(s) capturing all memory read access traces of the suspicious executable.

Detector. The cryptojacking malware and CryptojackingTrap listeners are both connected to the same cryptocurrency networks and malware mines over the blocks that are also visible to the listeners. The detector module is responsible for taking BL and RL from listeners and monitor respectively and correlating and deciding if the application under test is malware. The detailed algorithm of the detector will be explained in the rest of this paper in Section IV-D.

The proposed CryptojackingTrap solution has two possible architectures: asynchronous and synchronous. The *asynchronous architecture*, which is currently adopted and has been explained, allows the listeners and monitor modules to run independently, without forcing them to execute simultaneously. This approach provides more flexibility and does not compromise the efficiency of the system. On the other hand, the *synchronous architecture* requires all modules to run at the same time, and the detector algorithm is merged inside the monitor module. This approach eliminates the need to store block logs and memory read logs since the monitor/detector module is aware of blocks and memory traces in real-time.

Therefore, although the synchronous architecture is a possible architecture from the design point of view, it is not practical with implementation considerations. The current asynchronous architecture forces the least interference in suspicious executable normal execution and provides more flexibility.

The CryptojackingTrap architecture is designed to be highly extensible and adaptable to evolving industry trends by facilitating the developers to incorporate new cryptocurrencies without affecting the integrity of core modules. Additionally, the architecture is modular and each module is capable of utilizing different platforms as long as they adhere to the internal data interface.

C. The Sequence of CryptojackingTrap Processes

Fig. 3 illustrates the time sequence of interactions between the client and CryptojackingTrap modules. The green section shared between listeners and monitor modules indicates that data for all these modules must pertain to the same time frame. Since the history of cryptocurrency P2P network data is always available, the necessity of executing `startXListenr` before `startMonitor(i)` can be eliminated, and it suffices to cover debugging time in the listeners and execute `getXBL` after `getReadLog(i)` (X for currency names). Ultimately, presenting the entire log files to the detector using `DetectCryptojacking` provides a detection response, which its algorithm will be elaborated further in the subsequent section.

TABLE II
SUMMARY OF NOTATION FOR CRYPTOJACKINGTRAP.

Level	Symbol	Meaning
SO	Ω_c	Predictable bytes count in each cryptocurrency c
	Ξ	Min size of each acceptable SO's substring
HO	Π	Min percentage of SO's hash coverage that specifies one HO
	Δ	Max window of all SOs to specify one HO
MO	Ψ	Min number of HOs to specify one MO
	Θ	Max window of all HOs to specify one MO
CO	Γ	Min number of MOs to specify the CryptojackingTrap alert
	Λ	Max window of all MOs to specify the CryptojackingTrap alert

D. The Detection Module Algorithm

Cryptojacking must calculate a hash function repeatedly as the main process and because this function input size is bigger than the processor's registers, a processor can not retrieve it once and keep it for further calculations. Consequently, the processor has to retrieve this value continuously during mining. This repeated data retrieval persisted in RL and significant repetition of BL value in RL with a consistent timestamp of each record implies mining on the suspicious executables.

The hash function input value including the previous block hash is not expected to be retrieved at once and intact but it is retrieved fragmented and also fragment's size depends on each hash algorithm. To tackle the uncertainty of splitting the hash value into unknown parts with undetermined size, four levels of abstraction were introduced to organize these data and provide a significant probability to specify the correct alert of cryptojacking detection. The notations that are used in the algorithms and calculations are listed on TableII and organized based on the abstraction levels that will be described more in the remaining.

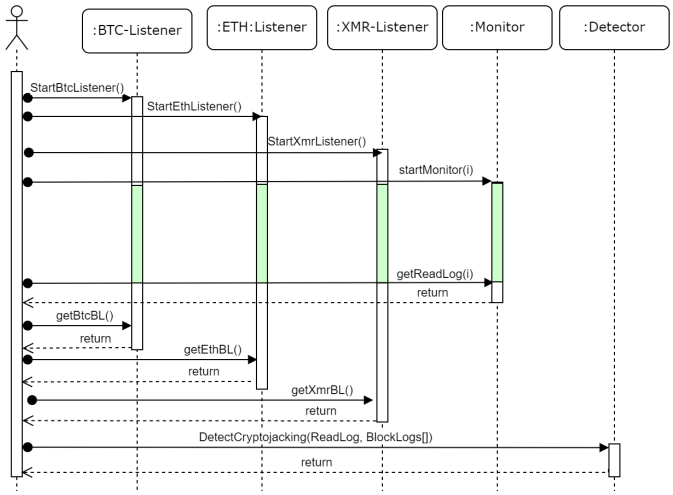

Fig. 3. Sequence diagram of CryptojackingTrap.

The levels of abstraction introduced by CryptojackingTrap are arranged from the lowest to the highest level, with each level building on the abstraction concept from the lower layer. As each level progresses, the criteria for true CryptojackingTrap detection becomes more stringent, resulting in higher detection precision. The examples provided in Fig. 4 visually illustrate these levels, aiming to help the reader develop an intuitive understanding of the algorithms discussed in this section.

Split Occurrence (SO). In the first level of detection abstraction, which is the lowest level, finding the substrings of the BL's hash values in the RL file is targeted. It is apparent from the statistics and probability basic calculations that the bigger the split substrings of hash value can appear in the RL file, the more probable it can be the indication that the executable file reads this specific hash value from memory. Consequently, in this level of abstraction, the detector searches for the substrings named splits with the following properties: Each split must be the largest feasible part of the hash that occurs in the RL file. The splits do not overlap with each other. The split sizes must be bigger than a threshold (Ξ), and they must appear exactly in the same order that they are in the hash value. In Fig. 4 example, the splits of the first cryptocurrency hash value are s1, s2, and s3, and the splits for the second cryptocurrency are s1 and s2.

Algorithm 1 Detector Algorithm: Find Split Occurrences

```
1:  procedure findSplitOccurrences(hash: String, readLog: File): SplitOccurrence[]
2:      mSplit ← findLCS(hash, readLog)                          ▷ main split
3:      if mSplit is null then
4:          return null
5:                                                  ▷ left side recursive call:
6:      lHash ← hash.subStr(0, mSplit.strStartIdx)
7:      lFile ← readLog.subFile(0, mSplit.rlStartIdxes[0])
8:      lSplits ← findSplitOccurrences(lHash, lFile)
9:                                                 ▷ right side recursive call:
10:     rHash ← hash.subStr( mSplit.strEndIdx, hash.size)
11:     rfile ← file.subFile(mSplit.rlEndIdxes[0], readLog.size)
12:     rSplits ← findSplitOccurrences(rHash, rfile)
13:     return new SplitOccurrence[] {lSplits,mSplit,rSplits}
14:
15: procedure findLCS(str: String, readLog: File): SplitOccurrence
16:     s ← new SplitOccurrence
17:     s.splitStr ← LCS (str, readLog)
18:     if s.splitStr < Ξ then
19:         return null
20:     s.strStartIdx ← the left index of splitStr in str
21:     s.strEndIdx ← the right index of splitStr in str
22:     s.rlStartIdxes[] ← find all left indexes of splitStr in readLog
23:     s.rlEndIdxes[] ← find all right indexes of splitStr in readLog
24:     return s
```

After finding these substrings, this procedure constructs a SplitOccurrence object that includes detailed information about this substring. This detailed information includes the

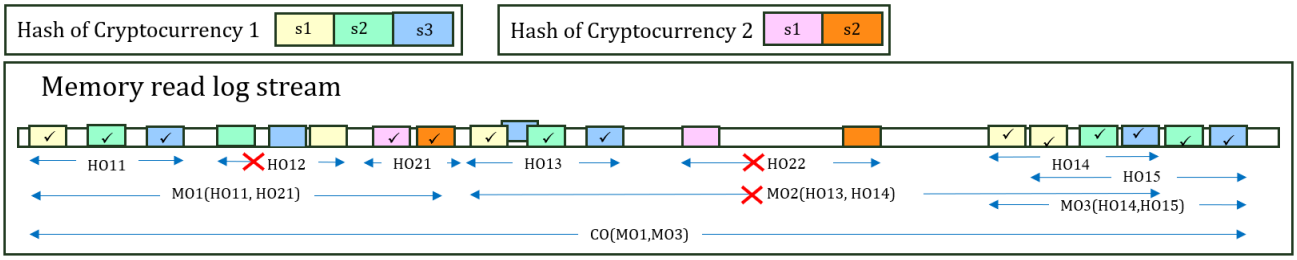

Fig. 4. Illustrative examples demonstrating levels of CryptojackingTrap detection abstraction.

substring content and indexes to address it in both the hash string and RL file. To find the biggest possible splits a greedy divide and conquer procedure is declared in Algorithm 1, named `findSplitOccurrences`. It starts by calling `findLCS` procedure on line 2 which is a pure string processing procedure that finds the *longest common substring (LCS)* shared between `hash` and `readLog` and terminates the procedure by returning `null` if found substring is less than Ξ on line 18 and 19. `SplitOccurrence` object includes other integer attributes: `strStartIdx` and `rlStartIdxes` are the first indexes of the firstly found `splitStr` in the `hash` (line 20) and `readLog` (line 22) respectively. `strEndIdx` and `rlEndIdxes` are the last indexes of the firstly found `splitStr` in the `hash`(line 21) and `readLog` (line 23) respectively. To make it clearer, `strEndIdx` minus `strStartIdx` is equal to `rlEndIdxes[i]` minus `rlStartIdxes[i]` is equal to `splitStr` size.

`subStr(integer fromIdx, integer toIdx)` is used on line 6 and 10 to return the substring of the callee string from the first index to the second index. `subFile(integer fromIdx, integer toIdx)` on line 7 and 11 similarly returns a new file containing the content of the callee file from the first index to the second index. consequently, `lHash` and `lFile` on line 6 and 7 are the left parts of `hash` and `readLog` respectively before the main split on them and `rHash` and `rFile` on line 10 and 11 are the right ones after the main split. `subFile` is a pseudocode to simplify its logic, but considering the performance concerns the same logic is implemented by passing indexes over one file. On line 13 a list including the left splits, main split, and right splits is returned as a result by preserving the order of split occurrences.

Hash Occurrence (HO). HO is the second level of detection abstraction indicates an attempt of reading a hash value from memory. Since an executable can not read a hash value from memory just by a single memory read access, HO encapsulates a list of splits that nondeterministically indicates one hash occurrence. HO guarantees a significant probability of correctness by enforcing criteria over splits selection specified in Algorithm 2.

The algorithm utilizes a temporary data structure called `section` (instantiated on line 10) to organize splits that correspond to a single HO. While each `SplitOccurrence` generated on line 2 denotes a split string and all its occurrences in `readLog`, a `section` refers to a specific split occurrence. Line 8 defines a list of sections that pertains to all splits and their splitIdxth occurrences, which could potentially indicate an HO if it satisfies the following criteria.

Algorithm 2 Detector Algorithm: Find Hash Occurrences

```
1:  procedure findHashOccurrences(hash: String, readLog: File): HashOccurrence[]
2:      sos ← findSplitOccurrences(hash, readLog)
3:      hos ← new HashOccurrence[]
4:      maxHOCount ← min size for rlStartIdxes lists per split in sos
5:      for splitIdx = 0; splitIdx ⩽ maxHOCount; splitIdx++ do
6:          ho ← new HashOccurrence()
7:          ho.hash ← hash
8:          sections ← new Section[]
9:          for so: sos do
10:             section ← new Section
11:             section.splitStr ← so.splitStr
12:             section.rlStartIdx ← so.rlStartIdxes[splitIdx]
13:             section.rlEndIdx ← so.rlEndIdxes[splitIdx]
14:             sections.add(section)
15:         if sections[sections.size-1].rlEndIdx -
16:         sections[0].rlStartIdx ⩽ Δ then
17:             sum = ∑_{i=0}^{sections.size−1} sections[i].splitStr.size
18:             hcp ← sum / hash.size * 100
19:             if hcp ≥ Π then
20:                 ho.sections ← sections
21:                 ho.rlStartIdx ← sections[0].rlStartIdx
22:                 ho.rlEndIdx ← sections[sections.size-1].rlEndIdx
23:                 hos.add(ho)
24:     return hos
```

The first criterion is that all splits within an HO must occur within a window of size Δ in `readLog`, as indicated on line 15 and 16. The second criterion is that concatenating the *splitStr* strings of the *sections* in the correct order must result in a string that is nearly identical to the input *hash* value. Specifically, the concatenated value can only differ by a few characters, which leads to the concept of *hash coverage percentage (HCP)*. The HCP is calculated as the sum of the sizes of all split strings in the HO's sections divided by the length of the hash and multiplied by 100. To ensure this criterion is met, the HCP for each HO is checked against a threshold value of Π on line 19.

In Fig. 4 examples, HO_{ij} represents the jth suggested hash occurrence for the ith cryptocurrency, with accepted splits indicated by check marks. HO11 is valid since it includes all ordered splits within an accepted window, while HO12 is invalid due to unordered splits. HO21 demonstrates interleaved hash occurrences, HO13 shows a selection of non-overlapping splits, HO22 rejects too-distant splits, and HO14 accepts all splits within an accepted window even though there might be extra valid splits in that window.

Mining Occurrence (MO). The third level of abstraction is dedicated to structuring a mining activity based on found HOs. The pseudocode for finding MOs from HOs is explained

in Algorithm 3. The algorithms benefit from object-oriented notions such as DetectorSetting.lstnSetting.currencies on line 3 to organize data based on supplier modules. Since the hash function must be repeatedly calculated in the cryptojacking algorithm, CryptojackingTrap is designed to be sensitive to the minimum count (Ψ) of detected HOs (line 30) in a limited window size of Θ (line 26). Initially findMiningOccurrences is called

In Fig. 4, MO1 is the initial valid mining occurrence considering $\Psi = 2$, encompassing HO11 and HO21 as valid hash occurrences. Notably, disparate cryptocurrency hash occurrences can collectively form an MO, curbing malware evasions across mining on varied platforms. MO2 is invalidated due to the distance between HO13 and HO14, exceeding Θ. By omitting HO13 and restarting the MO window from HO14, followed by HO15, it ended with the accepted MO3. This restarting of the window is done on line 37 in Algorithm 3.

Algorithm 3 Detector Algorithm: Find Mining Occurrences

```
1:  procedure findMiningOccurrences(s: DetectorSetting): MiningOccurrence[]
2:      hos ← new HashOccurrence[]
3:      for String c ∈ s.lstnSetting.currencies do
4:          t_start ← min(s.mntrSetting.RL.getAllTimestamps())
5:          t_end ← max(s.mntrSetting.RL.getAllTimestamps())
6:          for i_t = 0; i_t ⩽ s.lstnSetting[c].BL.size-1; i_t++ do
7:              t ← s.lstnSetting[c].BL[i_t].timestamp
8:              if t_start ⩽ t ⩽ t_end then
9:                  t̂ ← s.lstnSetting[c].BL[i_t + 1].timestamp
10:                 rl ← getSubRL(s.mntrSetting[c].RL, t, t̂)
11:                 h ← s.lstnSetting[c].BL[i_t].blockHash
12:                 hos.add(findHashOccurrences(h, rl))
13:     mos ← findMOsFromHOs(hos)
14:     return mos
15:
16: procedure findMOsFromHOs(hos: HashOccurrence[]): MiningOccurrence[]
17:     mos ← new MiningOccurrence[]
18:     for i = 0; i ⩽ hos.size-1; i++ do
19:         mo ← new MiningOccurrence()
20:         ho_start ← hos.get(i)
21:         selectedHOs ← new HashOccurrence[]
22:         selectedHOs.add(ho_start)
23:         mo.rlStartIdx ← ho_start.rlStartIdx
24:         for j = i+1; j ⩽ hos.size; j++ do
25:             ho_end ← hos.get(j)
26:             if ho_end.rlEndIdx-ho_start.rlStartIdx ⩽ Θ then
27:                                   ▷ collect HOs in a Θ-sized window
28:                 selectedHOs.add(ho_end)
29:                 mo.rlEndIdx ← ho_end.rlEndIdx
30:                 if selectedHOs.size ⩾ Ψ then
31:                     mo.setHOs(selectedHOs)
32:                     mos.add(mo)
33:                     i = j+1
34:                     break      ▷ continue with finding the next mo
35:                 else continue   ▷ add the next HO to selectedHOs
36:             else break      ▷ could not collect Ψ hos in a Θ-sized
37:                 window, start with the next HO as the first one
38:     return mos
39:
40: procedure getSubRL(srcRL: File, from: TimeStamp, to: TimeStamp): File
41:     return a new file including selected lines from RL file that
    their timestamp are between from and to timestamps
```

CryptojackingTrap Occurrence (CO). This is the fourth and highest level of abstraction used to determine if a suspicious executable is a cryptojacking malware, as described in Algorithm 4. The DetectorSetting input object (line 1) contains all the necessary data for detector execution, including lstnSetting and mntrSetting, which represent the complete data received from the listener and monitor modules, respectively.

Algorithm 4 Detector Algorithm: Is Cryptojacking

```
1:  procedure isCryptojacking(setting: DetectorSetting): Boolean
    ▷ returns true if finds Γ MOs in a Λ-sized window
2:      mos ← findMiningOccurrences(setting)
3:      for i = 0; i ⩽ mos.size-1; i++ do
4:          mo_start ← mos.get(i)
5:          acceptedMOs ← new MiningOccurrence[]
6:          acceptedMOs.add(mo_start)
7:          for j = i+1; j ⩽ mos.size-1; j++ do
8:              mo_end ← mos.get(j)
9:              if mo_end.rlEndIdx-mo_start.rlStartIdx ⩽ Λ then
10:                 acceptedMOs.add(mo_end)
11:                 if acceptedMOs.size ⩾ Γ then
12:                     print "Positive Alarm!" + acceptedMOs
13:                     return true
14:                 else continue  ▷ add the next MO to acceptedMOs
15:             else break              ▷ restart accepted MO window
16:     print "Negative Alarm!"
17:     return false
```

The isCryptojacking procedure output is a boolean value to indicate the detection result of cryptojacking activity in the input setting (line 1). This procedure retrieves all mining occurrences (line 2) and checks if there are at least Γ MOs (line 11) in a Λ-sized window (line 9). In Fig. 4, assuming Γ = 2 and counting MO1 and MO3 as the accepted mining occurrences, the algorithm ends to the final CryptojackingTrap alarm in the last layer of abstraction.

Although the algorithm elaboration in this section is self-sufficient, readers are facilitated to follow the implementation by using the same naming conventions simultaneously. Moreover, the algorithm serves as a seamless transition from the object-oriented declarative notation used in Java-based code to the concise abbreviations used in the formal evaluation section, which will be discussed in Section V-A.

E. CryptojackingTrap Implementation

This section provides technical details on the implementation of the CryptojackingTrap modules. The listeners are implemented in Java and support Bitcoin, Ethereum, and Monero cryptocurrencies. Bitcoin implementation has a user interface to specify the P2P nodes count while others are console-based applications. Despite language and design differences, the listeners produce output files in a uniform format. The monitor module is implemented using the Pin debugger software [15], described in [16]. The current experimental evaluation benefits from Pin 3.22 (February 28, 2022), with instrumentation coded in C++ using Visual Studio 2022. The output DLL (Dynamic Link Library) file generated by the instrumentation is fed into the Pin debugger to log the memory read access traces of the suspicious application. Respecting the monitor module's predefined file format, it is possible to substitute the Pin debugger with any off-the-shelf debugger. The detector is a Java-based project equipped with an interface, that facilitates users to specify the BL file(s) and cryptocurrencies of interest for scanning, as well as their corresponding RL file(s)

obtained from the respective listeners. By running the detector, users can trace the debug logs until the final positive or negative alert is announced. The project's entire codebase and resulting datasets are available for open science on GitHub. [1]. The project encompasses nearly 6.5K SLOC, and has a distinct repository for any module for independent versioning. These modules are monitor, listener-bitcoin, listener-ethereum, listener-monero, detector, util, and dataset. Further details on the implementation and file formats can be found in the comprehensive project documentation.

V. Evaluation

CryptojackingTrap is evaluated through two approaches. The first approach involves a mathematical evaluation to calculate the false positive rate (Section V-A), while the second approach focuses on calculating false positive and false negative rates using experimental evaluation approaches (Section V-B).

A. Mathematical Evaluation

This section quantitatively evaluates the accuracy of the CryptojackingTrap algorithm in detecting cryptojacking activities. To carry out these actions, the data and calculations of all modules (listener, monitor, and detector) are formalized and used to calculate the false positive rate. For the sake of consistency in this evaluation, 'blocks' will be used to refer to cryptocurrency blocks, irrespective of their type (e.g., Bitcoin, Ethereum, Monero, etc.).

Listener. The listener module maintains block log (BL) files for all supported cryptocurrencies (C). The BL for a cryptocurrency $c \in C$ is denoted as BL_c and formulated in Eq. (1). BL_c is as a finite chronologically ordered sequence of pairs (t_i, h_i), where t_i denotes the creation timestamp of the i^{th} block and h_i represents the corresponding block hash value. The counter i iterates through the recorded block log hashes within the block log file(s) specific to cryptocurrency c, starting at 1 and ending at BC_c, which represents the total block count within the BL_c file(s).

$$BL_c = \{(t_i, h_i) \mid i \in \{1, 2, 3, \ldots, BC_c\}\} \tag{1}$$

Monitor. The output read log (RL) of the monitor module is represented as a finite ordered sequence of (t_j,c_j) tuples in the following formula organized chronologically, with $j \in \{1, 2, 3, \ldots, RC\}$, where t_j represents the timestamp of the j^{th} memory read and c_j is the corresponding memory content. The value of RC ("read count") represents the number of memory reads recorded in the RL during the monitor module debugging time slot.

$$RL = \{(t_j, c_j) | j \in \{1, 2, 3, \ldots, RC\}\} \tag{2}$$

Detector. The detector module is responsible for processing the outputs of the listeners (Eq. 1) and monitor (Eq. 2) modules. The first step of the process formalization is to link memory read accesses to the current block on the cryptocurrency network at the precise time of the memory read. To achieve this objective, the Block Read Log (BRL_c), as defined in Eq. (3), is introduced. This is a combination of BL_c from Eq. (1) and RL from Eq. (2), jointed by the shared parameters of time t_i and t_j, respectively. Eq. (3) implies that the latest block hash of cryptocurrency c at the time of reading c_j by the suspicious executable was h_i. Since BRL_c tuples are defined according to each of the tuples in RL, the BRL_c list has the same size as RL file(s), and equals RC.

$$\begin{aligned} BRL_c = \ & \{(h_i, c_j) | \forall (t_j, c_j) \in RL, \exists (t_i, h_i) \in BL_c, \\ & t_i \leqslant t_j < t_{i+1})\} \end{aligned} \tag{3}$$

The second step of detector process formalization is breaking down c_j in BRL_c into its individual bytes and defining the ordered data list as described in Eq. (4) where D_c is data for cryptocurrency c and $\|$ notation is used for concatenation operation. Each pair (h_i, b_i) of D_c means that the suspicious executable reads b_i byte from memory and at a time the current block hash in the cryptocurrency network c is h_i and this list is ordered chronologically based on memory read access bytes that are preserved from the origin RL sequence. D_c is the sole input for formalizing the detector abstraction levels in the rest of this section, which was introduced in Section IV-D.

$$\begin{aligned} D_c = \{(h_i, b_i) | & (h_i, c_i) \in BRL_c, \\ & c_i = b_1 || b_2 || b_3 || \cdots \in \{0, 1\}^8\} \end{aligned} \tag{4}$$

The purpose of this section is to formalize a method for calculating the false positive rates of CryptojackingTrap using statistical mathematics. However, due to the high complexity of the algorithm, it is not possible to include all of the implementation details in the mathematical calculations. In order to address this complexity, the formalization process is simplified by respecting a certain *simplification principle*. The principle is that these simplifications must designed in a way that makes the detection criteria weaker, thereby increasing the detection rate and guaranteeing an increase in the false positive rate. If the false positive rate obtained from the simplified formalization is satisfactory, it can be safely assumed that the rate for the actual algorithm and implementation, which has a lower false positive rate, will also be satisfactory. One of the adopted simplifications is to ignore the minimum size of splits (Ξ) in the first level of abstraction or in other words, it is assumed that $\Xi = 0$. Decreasing the minimum size of splits increases the possibility of acceptance of random and incorrect data that do not originate from the malware memory access, and consequently, it increases the false positive rate and is compatible with the described simplification principle. Other assumptions are $\Pi = 100$, which means full hash coverage percentage in any hash occurrence. Assuming $\Psi = 2$ and $\Gamma = 1$, Λ=0 simplifies the question of finding two hash occurrences to notify the detection. Since this assumption makes the detection criteria weaker and increases the false positive rate, it is consistent with the defined simplification principle.

The second level of abstraction is the hash occurrence, which is denoted as HO_c in Eq. (5). HO_c is a sequence of hash occurrences for cryptocurrency c where each hash occurrence is represented as a Ω_c-tuple of hash-byte pairs from D_c

[1]https://github.com/CryptojackingTrap/CryptojackingTrap

(Line 1 and first statement of line 2). K_{i+1} is not necessarily equal to $(K_i + 1)$ (Last statement of line 2), which means the hash-byte tuples in each hash occurrence can be interrupted by other memory reads, produced by other functionality on the suspicious executable except hash calculations. However, these interruptions must occur within a Δ-sized window in the RL, which is expressed by $k_{\Omega_c} - k_1 \leqslant \Delta$ in line 3. Note that, the indexes are originated from D_c and consequently are an indicator of its byte location in the RL file(s). It is evident that Δ is greater than Ω_c because there are Ω_c tuples in a Δ-sized window. All hash values in one hash occurrence tuple are identical and equal to the concatenation of all byte values in that tuple while preserving their order (Line 4).

$$\begin{aligned} HO_c = \{&((h_{k_1}, b_{k_1}), (h_{k_2}, b_{k_2}), \ldots, (h_{k_{\Omega_c}}, b_{k_{\Omega_c}}))| \\ &\forall i, 1 \leqslant i \leqslant \Omega, (h_{k_i}, b_{k_i}) \in D_c, k_i + 1 \leqslant k_{i+1}, \\ &b_{k_i} \in \{0,1\}^8, k_{\Omega_c} - k_1 \leqslant \Delta \\ &h_{k_i} = h_{k_j}, h_{k_i} = b_{k_1}||b_{k_2}||b_{k_3}||\ldots||b_{k_{\Omega_c}}\} \end{aligned} \tag{5}$$

The next abstraction level, mining occurrence (MO), is formalized in Eq. (6).

$$\begin{aligned} MO = \{&(ho_{k_1}, \ldots, ho_{k_\Psi})| \\ &\forall i, 1 \leqslant i \leqslant \Psi, \exists c, ho_{k_i} \in HO_c, k_\Psi - k_1 \leqslant \Theta\} \end{aligned} \tag{6}$$

and the occurrence of CryptojackingTrap (CO) using the formalization presented in Eq. (7).

$$CO = \{(co_{k_1}, \ldots, co_{k_\Gamma})|co_{k_i} \in MO, k_\Gamma - k_1 \leqslant \Lambda\} \tag{7}$$

The ultimate CryptojackingTrap detection is then specified in (8) using these formalizations.

$$|CO| > 0 \Rightarrow CryptojackingTrap\ alarm \tag{8}$$

Using the formalized data and processes presented in this section, the false positive rate of the proposed algorithm can be calculated. This metric can be represented mathematically as the probability of obtaining the Eq. (8) event within a randomly generated RL file distribution. To calculate the false positive rate, a probability question arises that must first be discussed in a general format outside of the context of CryptojackingTrap, and then applied to the CryptojackingTrap context. Specifically, the contextualized question is as follows: What is the probability of the following event in a random sample space D_c? The event is defined as "finding Ω_c members of a predefined list in their original order in a Δ-sized sub-list of D_c, as well as finding these Ω_c members in another Δ-sized sub-list in D_c, where the distance between the first item of the first subset and the last item of the second subset is less than or equal to Θ."

To address the above-mentioned probability question, the *inclusion-exclusion principle* is employed, as expressed by the Eq. (9). This principle computes the number of surjective functions from set A, which consists of k elements, to set B, consisting of n elements. Eq. (9) also provides the number of distributing k distinct objects in n places, ensuring that all k objects are present and each of the n places receives at least one object and is non-empty.

$$T(k, n) = \sum_{i=0}^{n} \binom{n}{i} (-1)^i (n - i)^k \tag{9}$$

Thus, the probability of having Ω_c members in origin order in Δ series of sample space is calculated as below:

$$HOC_\Delta = \binom{\Omega_c}{1} \times T(\Delta - 1, \Omega_c - 1) \tag{10}$$

As a consequence, the count of Mining Occurrence is calculated as

$$MOC = (RC - \Theta) \times (HOC_\Delta)^2 \times \binom{\Theta - 1}{1}. \tag{11}$$

Eq.(5) formalizes hash occurrences, while Eq.(10) calculates the count of hash occurrence events in an RL file containing RC records. As a result, the probability of hash occurrence events in a random sample space RL with a size of RC can be computed using the following equation:

$$FP = \frac{MOC}{2^{RC}}. \tag{12}$$

Since Eq. (12) formulates a probability of alerting CryptojackingTrap in a random file, it represents the false positive of the CryptojackingTrap. Referring to Section IV-A, the previous block hash is used as the predictable field in the current implementation over Bitcoin, Monero, and Ethereum, and consequently, its variable is initiated as Ω_c = 32. Other variables are initiated based on the executions tuning over the implementation as Δ = 48, Θ = 1,000,000, and RC = 40,000,000 by scanning the suspicious executable file for 5 minutes. By substituting these values into Eq. (12) the false positive rate of CryptojackingTrapis calculated equal to 10^{-20}.

B. Experimental Evaluation

While the proposed solution undergoes evaluation using mathematical methods, assessing it with datasets offers a more comprehensive benchmark. However, obtaining authentic malware datasets adhering to policies presents significant limitations and challenges. The datasets organized before 2019, as detailed in [17], have become obsolete for benchmarking following the shutdown of Coinhive in March 2019. This investigation discovered that 99% of websites ceased cryptojacking activities, while 1% executed eight distinct mining scripts, revealing 632 potential cryptojacking sites upon script tracking. Given the dataset's lack of public accessibility, an additional investigation utilizing the same approach led to the identification of 130 websites. Nevertheless, none of these instances led to any negligible change in CPU usage, suggesting the inactive existence of these eight scripts on those websites. Given our emphasis on active malware, datasets featuring inactive scripts lack validity. Consequently, this study focused on diverse non-malware realistic datasets to strengthen the experimental evaluation of the method.

The Java code used to generate the files referenced in the evaluation, along with detailed testing results, is available in this project's dataset repository. The experiments are conducted on a laptop running Microsoft Windows 11 Pro

operating system, version 10.0.22621 Build 22621, with a 64-bit based PC system type. The laptop had an 11th Generation Intel(R) Core(TM) i5-1135G7 processor with four cores and eight logical processors, running at 2.40GHz. It was equipped with 16.0 GB of installed physical memory (RAM), with a total physical memory of 15.7 GB, and 2.36 GB of available physical memory. Additionally, it had 24.2 GB of total virtual memory with 3.25 GB available virtual memory, and a page file space of 8.50 GB. The evaluation time window targeted for any single test is on average one minute of monitoring that ends with a file of two million lines and around 80 MB. The precise values for each test are available in any test data file.

Table III provides more detailed information for the following experiments (1, 2, and 4) about the split occurrences that were found in each file that ultimately failed to meet the requirements for the accepted hash occurrence. Due to lack of space in Table III, the column names are shortened and here the complete meaning of each column is specified. The experiment information is organized into two main sections (randomly generated data and benign non-miner applications). It follows with the degree of randomization or application name in the "Name" column. Then the detector module's average execution time is specified. Coverage is the HCP defined in IV-D. The next column is the average split counts that are found for each test. It is followed by the maximum split size and average split size. The next section of columns is dedicated to specifying the reasons for discarding the split occurrences as 1) The splits that are not found on the read log file, 2) The splits with a size less than acceptable threshold size (Ξ), and 3) The valid splits that are discarded in calculating the hash occurrence because they are too far from the preceding splits and exceeds the hash occurrence window threshold (Δ). These three columns represent percentages indicating the proportion of splits that failed due to any declared reasons. Since no hash occurrence was detected in any records within this table, the mining process was skipped in all of these instances.

1) Randomly Generated Dataset Evaluation.

To calculate the false positive rate of the algorithm, testing was conducted using a random dataset of memory read logs and random 32-byte hex hash values as the monitoring data. The randomly generated monitoring file is a 2,000,000 lines file, where each line has random memory read traces, including random size and content. The test is a 100-round test, and each round uses a new random hash and the same mentioned read log file. In each round no cryptojacking activity was detected, demonstrating the zero false positive of the detector module. This negative detection is a very strong detector that will be explained more in the following. As discussed in Section IV-D CryptojackingTrap detector employs a multi-level abstraction to iteratively search for patterns of split, hash, mining, and CryptojackingTrap occurrences in increasing order of restrictiveness and if any of them are found, detector imposes more restrictive criteria to search the next level and the detector comes with the positive answer if the last level is found. In this experiment, in addition to the negative response for detection, there were no occurrences of mining or even hash patterns found. This underscores the effectiveness of the CryptojackingTrap detector in accurately identifying the absence of cryptojacking activity. In Table III, the first row labelled "100% Random" is the result of this dataset category.

The accepted split occurrences only covered 27.91% of the hash value size, which falls significantly short of the minimum hash coverage requirement of 80%. Consequently, the generated split occurrences were not substantial enough to create a hash occurrence that met the minimum coverage requirement.

2) Benign Non-Miner Applications Evaluation.

CryptojackingTrap is evaluated in this section by testing on the benign non-miner highly used applications listed in Table III. The reported numbers are the average of six different test results for each application. Likewise, randomly generated dataset, all the tests conducted in this category resulted in a strong negative alert which means a zero false positive rate.

3) Benign Miners Evaluation.

Due to the open-source nature of most crypto mining programs, attackers can modify the code base to create their variants. For instance, attack instances including CryptoSink, RubyMiner, JenkinsMiner and Graboid utilize the XMRig benign open source miner [18] [19]. As a result, the evaluation of CryptojackingTrap's true positive rate is conducted by executing it alongside the benign miners on Bitcoin, Ethereum, and Monero networks in the remaining of the current section.

Bitcoin used to be a profitable cryptocurrency for attackers in recent years that was the target of studies [5]. Bitcoin CPU mining is not beneficial anymore in recent years and hence is not supported by many miner applications. For example, DiabloMiner [20] is not valid any longer, and Bfg-Miner [21] and CgMiner [22], which used to be the most famous Bitcoin miners, removed Bitcoin mining from their default configurations. This unprofitability is due to more electricity consumption per hash calculation than GPU, FPGA, or ASIC-based mining. In the scope of this study, Bitcoin mining is included in evaluations because Cryptojacking does not raise electricity consumption concerns for the attacker, as it utilizes victims' machines. Consequently, the current section analyzes Bitcoin CPU mining using CpuMiner [23], resulting in successful detection with a zero false-negative rate.

MinerGate [24] was selected as an Ethereum mining tool for the tests. This miner autonomously optimizes profit by selecting cryptocurrency types. While there is an option to disable certain currencies, the inability to exclude Monero makes it unsuitable for Ethereum testing. Additionally, WinEth [25], another Ethereum mining software, is incompatible with the evaluation system's video card. MinerGate [24], intended for Ethereum, was also utilized for Monero mining, and CryptojackingTrap successfully detected it with a zero false-negative rate.

4) Randomized Benign Miner Dataset Evaluation.

This dataset is created by randomizing an original RL file, created by monitoring MinerGate which mines Monero. Randomization evaluation is conducted by different "probabilities" that are (1, 5, 10, 15, ..., 85). For example, 20 percent randomization means all the lines of the files are replaced with a randomly generated line (random valid size and value) with 20% probability that ends up changing almost 20% lines of

TABLE III
RANDOM GENERATED (UPPER) AND BENIGN NON-MINER APPLICATIONS (LOWER) TEST RESULTS.

	Name	Time	Valid Split Occurrences Information				Discarded Split Occurrences Error Types		
			Coverage	Splits Count	Max Split Size	Avg Split Size	MISSED	TOO_SMALL	TOO_FAR
Random Generated	100% Random	35.55 s	27.91 %	17.53	5.11	3.63	0.91 %	34.54 %	37.47 %
	85% Random	36.34 s	60 %	13	18	6.0	0 %	30.77 %	23.07 %
	50% Random	49.57 s	68.5 %	8	18	12.5	0 %	37.5 %	18.75 %
	15% Random	47.92 s	75 %	8	18	12.0	25 %	12.5 %	12.5 %
Benign Non-Miners	Adobe Acrobat DC	37.03 s	21.60 %	32.33	4.15	3.00	8.76 %	62.37 %	16.13 %
	Microsoft Calculator	31.54 s	21.93	28.50	3.78	3.00	7.02 %	57.31 %	20.37 %
	Microsoft Word	30.87 s	25.17	30.17	4.00	3.00	8.29 %	57.46 %	17.68 %
	Nodepad++	24.76 s	19.76	27.83	3.30	3.00	11.38 %	53.29 %	20.26 %
	Photos	26.02 s	20.12	30.00	3.79	3.10	12.22 %	57.22 %	17.04 %

the origin file. Table III lists 3 records amongst 18 conducted experiments that resulted in negative alarms. The results show that CryptojackingTrap continues to give positive alarms till 5% randomization and stops detecting herein after and till 40% randomization detects HO but no MO that leads to a Negative alarm.

5) Expanded Benign Miner Dataset Evaluation.
The difference between this dataset and the preceding one is that this dataset does not create distortion in patterns and the target is to make them more sparse and spread random data between each two lines. This approach is a simulation of a miner that does not dedicate its all process to mining and decreases its hash rate for evasion purposes. The metric used in creating this dataset is how much "$times$" the data is targeted to expand. For example, using $times = 10$ expand the origin file ten times and consequently for every line of the origin file $times - 1$ random lines are added after each line. The evaluation of $times = 2, 3, 4, ..., 12$ concluded that CryptojackingTrap is resilient to detect until ten times of hash rate reduction.

C. Discussion

This section presents a summary of the evaluations conducted in the preceding subsections, focusing on the false positive and false negative rates of the detector algorithm. The analysis encompasses both mathematical calculations and experimental assessments, which are divided into five subsections. The mathematical subsection establishes an almost negligible false positive rate (10^{-20}). Subsequent assessments, including tests on randomly generated datasets and benign non-miner applications, consistently affirm the zero false positive rate, corroborated by mathematical results. Evaluations of benign miners also indicate a zero false negative rate among the tested miners. Moreover, the algorithm underwent rigorous evaluation in unconventional and challenging scenarios, specifically through assessments using randomized benign miner datasets and expanded benign miner datasets. The former demonstrates the robustness of the approach, displaying a resilient zero false negative rate even under conditions of up to 40% randomization of memory read content. The latter observations confirm that the algorithm maintains a zero false positive rate while accommodating up to a tenfold reduction in malware mining rates—an evasion technique used by *cryptj* malware that is not commonly addressed in many detection approaches, such as [26].

VI. RELATED WORK

This section provides an overview of the most pertinent and up-to-date research on cryptojacking malware detection and explains how CryptojackingTrap outperforms these methods in terms of resilience and precision.

In currently available cryptojacking detection approaches, certain assumptions are made that are not universally correct, resulting in a lack of resilience against malware evasion mechanisms and reduced precision. In the following the comprehensive classification of evasion techniques and the studies that are not resilient against them are listed. The first flawed assumption is that cryptojacking exhausts the victim's CPU resources [27]. However, studies have demonstrated that attackers use *hash rate reduction* by ten times to evade detection by the victim. For example, Gangwal and Conti introduced an innovative method utilizing smartphone magnetic sensors to profile processor magnetic field emissions across various mining algorithms, employing sophisticated machine learning techniques [26]. Their experimentation, conducted on two laptops, yielded notable results with an average precision exceeding 88% and an average F1 score of 87%. However, it is important to note a limitation of this approach: it may encounter challenges in accurately detecting hash rate reduction scenarios, and even when operating at full hash rate capacity, it does not ensure a guaranteed detection rate.

The second wrong assumption is that cryptojacking web pages consistently obtain exact keywords as malicious payload signatures [28]. Nonetheless, researchers have revealed *payload hiding mechanisms* of the attackers and presented counter examples [9]. The third CryptojackingTrap represents the first resilient solution against all these evasion mechanisms, consistently outperforming them in terms of resiliency and precision.

Konoth et al. used cryptocurrency mining features to propose MineSweeper for detecting cryptojacking [6]. They refer to the cryptojacking attacks as drive-by mining attacks. While MineSweeper uses features such as "Cryptomining Code," "CPU load as a side effect," and "mining pool communication," these features are all susceptible to bypassing by the malware. For example, the use of specific patterns in the code, such as "coinhive.min.js," is not mandatory, and the

malware can establish its coin-mining service with unknown names. The CPU load can also be hidden by reducing the rate and allocating the remaining processing capacity to other malicious activities. Additionally, the mining pool communication can be concealed by using a proxy for communicating with miners and encrypting botnet communication. Therefore, MineSweeper's collected data is vulnerable to invalidation and evasion. All these diversity are discussed that can not affect the low-level activity of the miner that ends in valid detection by CryptojackingTrap.

The research in [5] proposed BotcoinTrap, which benefits from a new feature of botcoins, which is the memory access trace partial predictability feature. This method is resilient to obfuscation like the previous study [6], but in contrast with MineSwipper, BotcoinTrap can detect botcoins in terms of executable files or running processes. This approach can not be bypassed by changing general botnet features, including C&C protocol features. BotcoinTrap concerns solely malware that mines Bitcoin, which was the most used cryptocurrency at the time of research, 2018. Due to the cryptocurrency ecosystem dynamicity, this approach is not currently sufficient. CryptojackingTrap can detect cryptojacking activity on any cryptocurrency network. Another limitation of BotcoinTrap, which is covered in the current research, is the lack of precision metrics calculation for the proposed algorithm, neither formally nor with empirical proof.

The first opcode analysis attitude toward detecting crypto-miners was conducted by Carlin et al. [29]. This study analyzed opcode dynamically to detect mining activity over suspicious websites. Although opcode analysis is one of the well-known malware detection methods, it suffers intrinsically from numerous intrinsic limitations. The first limitation is that polymorphic malware can change its code structure at runtime, which makes it difficult to detect using opcode analysis. The malware can generate different opcodes each time it executes, which makes it challenging to develop reliable signatures for detection. The second limitation is obfuscation to hide the malicious code's true purpose. This can include techniques like packing, encryption, and code obfuscation. These techniques can make the opcode analysis approach less effective since it can be challenging to identify malicious behavior from the obfuscated code. Although this approach is skippable by polymorphic malware and obfuscation, CryptojackingTrap relying solely on non-hideable characteristics of low-level memory contents can conquer all these evasions.

Compared to the above methods, this proposed method focuses on detecting the hash function execution over cryptocurrency data using low-level memory access tracing. This approach is not dependent on function signature, code features, or hard-coded data, making it resistant to obfuscation techniques. Moreover, it is immune to diversity in botnet network protocols such as pool URL, protocol type, and C&C encryption, making it an efficient detection tool. The mathematical evaluation shows that CryptojackingTrap achieves a false positive rate of 10^{-20}, which is a new record for the precise detection of cryptojacking malware.

In conclusion, several methods have been proposed to detect cryptojacking malware. While some of these methods have achieved high accuracy, they suffer from a high false positive rate or are dependent on specific features of the malware. This proposed method focuses on detecting the hash function execution over cryptocurrency data using low-level memory access tracing, making it resistant to obfuscation techniques and immune to diversity in botnet network protocols. The evaluation shows that the method achieves high accuracy and a low false positive rate, making it an efficient detection tool for cryptojacking malware.

VII. Conclusions

In this paper, CryptojackingTrap, a novel and robust technique for detecting cryptocurrency mining activities in suspicious applications has been presented. This technique encompasses not only executable files but also operating system processes and websites. The approach focuses on the success phase of malware and tracing the low-level memory access of malware and predicts the data that miners must access. The predictable data is obtained from the cryptocurrency network to detect mining activities. Importantly, this method does not rely on function signature, code features, or hardcoded data, rendering it resistant to obfuscation techniques. Furthermore, it demonstrates immunity to diversity in botnet network protocols such as pool URL, protocol type, and C&C encryption, making it an efficient detection tool.

The dynamic ecosystem of cryptocurrencies in terms of new cryptocurrency birth or changing cryptocurrencies inclusion in malware does not jeopardize the functionality of CryptojackingTrap. The tool currently supports the detection of profitable cryptocurrencies of Bitcoin, Ethereum, and Monero mining activities and can be easily extended to include other cryptocurrencies by adding a listener extension with no change in its core modules, making it scalable and flexible.

Evaluation results confirm that CryptojackingTrap achieves high accuracy and a low false positive rate in detecting stealthy crypto-miners. The mathematical evaluation of its false positive rate yields an exceptionally low value of 10^{-20}. Additionally, rigorous testing over randomly generated benign and malicious samples validates the effectiveness of this solution. This highlights the importance of cryptojacking detection in modern cyber-security, where attackers continuously develop new and sophisticated methods to evade detection.

Since the main advantage of this proposed method is the resilience against most evasion techniques, it inherently requires processing that is not recommended to be carried out as a life background process. This approach mainly is useful to find new families of malware, which can then be formulated as lightweight signatures in the final user's security products. Furthermore, it is suggested that CryptojackingTrap can be beneficial for cyber-security products aimed at identifying zero-day attacks.

This research work includes the implementation of 6.5K SLOC, released as open source, enabling researchers to extend upon both the approach and the benchmarks according to their ideas. For future work, it is proposed that branches of similar size on blockchains be included since CryptojackingTrap works with the largest blockchain branch.

In conclusion, CryptojackingTrap emerges as a creative and resilient detection technique that provides a reliable solution for detecting cryptocurrency mining activities. Its robustness, extendability, and flexibility make it an essential tool for contemporary cyber-security.

ACKNOWLEDGMENT

Appreciation is extended to the anonymous reviewers for their invaluable contributions that significantly enhanced the paper's quality. Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Commission. Neither the European Union nor the granting authority can be held responsible for them. Prof. Marco Roveri is partially funded by HE - CROSSCON - GA 101070537.

REFERENCES

[1] H. Kettani and P. Wainwright, "On the top threats to cyber systems," in *2019 IEEE 2nd International Conference on Information and Computer Technologies (ICICT)*, 2019, pp. 175–179. [Online]. Available: https://doi.org/10.1109/INFOCT.2019.8711324

[2] D. Y. Huang, H. Dharmdasani, S. Meiklejohn, V. Dave, C. Grier, D. McCoy, S. Savage, N. Weaver, A. C. Snoeren, and K. Levchenko, "Botcoin: Monetizing stolen cycles." in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014, pp. 1–16. [Online]. Available: https://www.ndss-symposium.org/ndss2014/botcoin-monetizing-stolen-cycles

[3] S. Khattak, N. R. Ramay, K. R. Khan, A. A. Syed, and S. A. Khayam, "A taxonomy of botnet behavior, detection, and defense," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 898–924, 2014. [Online]. Available: https://doi.org/10.1109/SURV.2013.091213.00134

[4] R. Naraine, "Researchers find malware rigged with bitcoin miner," 2011. [Online]. Available: https://www.zdnet.com/article/researchers-find-malware-rigged-with-bitcoin-miner/

[5] A. Zareh and H. Shahriari, "Botcointrap: Detection of bitcoin miner botnet using host based approach," in *2018 15th International ISC (Iranian Society of Cryptology) Conference on Information Security and Cryptology (ISCISC)*, 2018, pp. 1–6. [Online]. Available: https://doi.org/10.1109/ISCISC.2018.8546867

[6] R. K. Konoth, E. Vineti, V. Moonsamy, M. Lindorfer, C. Kruegel, H. Bos, and G. 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*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1714–1730. [Online]. Available: https://doi.org/10.1145/3243734.3243858

[7] D. Hodick and A. Sievers, "On the mechanism of trap closure of venus flytrap (dionaea muscipula ellis)," *Planta*, vol. 179, pp. 32–42, 1989, publisher: Springer. [Online]. Available: https://doi.org/10.1007/BF00395768

[8] A. Zimba, Z. Wang, and M. Mulenga, "Cryptojacking injection: A paradigm shift to cryptocurrency-based web-centric internet attacks," *Journal of Organizational Computing and Electronic Commerce*, vol. 29, no. 1, pp. 40–59, 2019. [Online]. Available: https://doi.org/10.1080/10919392.2019.1552747

[9] 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*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1701–1713. [Online]. Available: https://doi.org/10.1145/3243734.3243840

[10] M. Musch, C. Wressnegger, M. Johns, and K. Rieck, "Thieves in the browser: Web-based cryptojacking in the wild," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ser. ARES '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3339252.3339261

[11] R. A. Rodríguez-Gómez, G. Maciá-Fernández, and P. García-Teodoro, "Survey and taxonomy of botnet research through life-cycle," *ACM Comput. Surv.*, vol. 45, no. 4, aug 2013. [Online]. Available: https://doi.org/10.1145/2501654.2501659

[12] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 421–430. [Online]. Available: https://doi.org/10.1109/ACSAC.2007.21

[13] R. Tahir, M. Huzaifa, A. Das, M. Ahmad, C. Gunter, F. Zaffar, M. Caesar, and N. Borisov, "Mining on someone else's dime: Mitigating covert mining operations in clouds and enterprises," in *Research in Attacks, Intrusions, and Defenses*, M. Dacier, M. Bailey, M. Polychronakis, and M. Antonakakis, Eds. Springer International Publishing, 2017, pp. 287–310. [Online]. Available: https://doi.org/10.1007/978-3-319-66332-6_13

[14] B. Authors, "Blockchain blocks version - bitcoinity.org," 2023. [Online]. Available: https://data.bitcoinity.org/bitcoin/block_version

[15] "Pin - a dynamic binary instrumentation tool," 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: https://doi.org/10.1145/1065010.1065034

[17] S. Varlioglu, B. Gonen, M. Ozer, and M. Bastug, "Is cryptojacking dead after coinhive shutdown?" in *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, 2020, pp. 385–389. [Online]. Available: https://doi.org/10.1109/ICICT50521.2020.00068

[18] K. Jayasinghe and G. Poravi, "A survey of attack instances of cryptojacking targeting cloud infrastructure," in *Proceedings of the 2020 2nd Asia Pacific Information Technology Conference*, ser. APIT '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 100–107. [Online]. Available: https://doi.org/10.1145/3379310.3379323

[19] xmrig, "XMRig," 2017, original-date: 2017-04-15T05:57:53Z. [Online]. Available: https://github.com/xmrig/xmrig

[20] P. McFarland, "DiabloMiner: OpenCL miner for bitcoin," 2018, original-date: 2010-11-06T14:56:14Z. [Online]. Available: https://github.com/Diablo-D3/DiabloMiner

[21] L. Dashjr, "luke-jr/bfgminer," 2023. [Online]. Available: https://github.com/luke-jr/bfgminer

[22] C. Kolivas, "CGMiner 3.7.2 download (windows 10) AMD, doge [2023]," 2023. [Online]. Available: https://cgminer.info/

[23] C. M. Contributors, "pooler/cpuminer," original-date: 2011-12-04T19:26:44Z. [Online]. Available: https://github.com/pooler/cpuminer

[24] T. M. Authors, "Smart multicurrency mining pool & 1-click GUI miner," 2023. [Online]. Available: https://minergate.com/

[25] T. W. Authors, "Easy graphical ethereum mining software," 2023. [Online]. Available: https://wineth.net/

[26] A. Gangwal and M. Conti, "Cryptomining cannot change its spots: Detecting covert cryptomining using magnetic side-channel," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 1630–1639, 2020.

[27] D. Goodin, "Cryptojacking craze that drains your CPU now done by 2,500 sites," 2017. [Online]. Available: https://arstechnica.com/information-technology/2017/11/drive-by-cryptomining-that-drains-cpus-picks-up-steam-with-aid-of-2500-sites/

[28] S. Eskandari, A. Leoutsarakos, T. Mursch, and J. Clark, "A first look at browser-based cryptojacking," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2018, pp. 58–66. [Online]. Available: https://doi.org/10.1109/EuroSPW.2018.00014

[29] D. Carlin, P. O'Kane, S. Sezer, and J. Burgess, "Detecting cryptomining using dynamic analysis," in *2018 16th Annual Conference on Privacy, Security and Trust (PST)*, 2018, pp. 1–6. [Online]. Available: https://doi.org/10.1109/PST.2018.8514167