Robbery on DevOps: Understanding and Mitigating Illicit Cryptomining on Continuous Integration Service Platforms

***Abstract*—The recent wave of in-browser cryptojacking has ebbed away, due to the new updates of mainstream cryptocurrencies, which demand the level of mining resources browsers cannot afford. As replacements, resource-rich, loosely protected free Internet services, such as Continuous Integration (CI) platforms, have become attractive targets. In this paper, we report the study on a systematic study on real-world illicit cryptomining on public CI platforms (called *Cijacking*)that for the first time, discovers real-world attack instances on these platforms. Unlike in-browser cryptojacking, Cijacks masquerade as CI tasks jobs and are therefore more difficult to detect, since legitimate CI workflows such as container image building and testing also entail intensive computing. In our research, we leveraged the critical mining information the adversary has to specify, such as wallet addresses and mining pool domains, to recover the attack traces from GitHub repositories and the log files on CI platforms, leading to the discovery of 1,974 Cijacking instances, 30 campaigns across 12 different cryptocurrencies on 11 mainstream CI platforms. Further, our study unveils the evolution of attack strategies, in response to the protection put in place by the platforms, the duration of the mining tasks jobs (as long as 33 months), and their lifecycle. Further discovered is the revenue of the attack, over $20,000 per month.**

Since robust detection of cryptojacking is known to be hard, we developed a novel technique, called Cijitter, to strategically inject delays to the execution of a CI workflow to disproportionally penalize the mining tasks jobs that need to work on a series of jobs tasks under time constraints. Our analysis and evaluation, as conducted on both micro-benchmarks benchmarks and common CI workflows, shows jobs, show that our approach substantially suppresses the miner's revenues, rendering them unprofitable, but only has a small impact small impacts on the performance of the CI platform (below 5% throughput loss) and legitimate users (over 90% of tasks with CI jobs and developer productivity (94.3% of CI jobs see a less than 10% delay).

I. Introduction

The gold rush of crytocurrencies cryptocurrencies has attracted millions of digital miners. Among them are cybercriminals seeking wealth at others' expense through *cryptojacking*, a crime in which computing resources of compromised hosts or web browsers are abused for unauthorized cryptomining. Particularly, malicious actors are reported to deploy JavaScript miners through vulnerable or malicious websites so those visiting the sites automatically run the scripts in their browsers, unwittingly committing resources to serve the attacker's cause [59]. With thousands of such cryptojacking sites discovered and million dollar revenues generated from the stolen resources [55], recent updates of the mining algorithms underlying mainstream cryptocurrencies like Monero [62], however, start to render such in-browser cryptojacking ineffective, due to the requirement of gigabytes of memory for bootstrapping the mining computation, which cannot be afforded by browsers [54].

In response to this change, cybercriminals are have been reported to look for new targets – resource-rich , but less protected public services, such as platforms for software development and IT operation (*DevOps*).

Crytojacking CI platforms. Today's DevOps platforms support software development mainly through *continuous integration* (*CI*) and *continuous delivery* (*CD*). Commercial CI/CD services such as Travis-CI [34], CircleCI [10], Azure-pipeline [3] have already been integrated into the development workflows of over a million Github projects, helping their platforms [2], [33] in particular. DevOps platforms help developers build, test (the CI *continuous integration*, i.e., *CI*, step) and deploy (the CD *continuous delivery*, i.e., *CD*, step) applications. Among these steps, CI entails intensive computation , for building a project's Dockerfile as into a Docker image and further testing the image using the developers' scripts. Although such building and testing platforms are meant to serve their paid users, they also commit a significant amount of computing resources to the development tasks submitted by unpaid users. As an example, CircleCI offers a 14 days free trial that runs a Github project on a system with 4 vCPU, 8 GB memory. This level of computing support makes the CI platforms step and its associated platforms (e.g., TravisCI [34], CircleCI [10], Wercker [36]) appealing targets for cryptojacking, particularly for executing the mining algorithms like RandomX used by Monero [62] , which need gigabytes of memory to start with. . However, little has been done so far to understand whether indeed these platforms have been abused for cryptominingsystematically discover and analyze such trending cryptojacking activities, not to mention any effort to mitigate the security risk they are exposed to.

Finding Cryptojacking on the CI platforms. In our research, we performed the *first* a systematic analysis of cryptojacking risks on 23 popular CI platforms, which not only confirms the presence of such malicious activities but also brings to light their significant impacts never known before. More specifically, to discover the attempts to abuse a CI service for cryptomining, which we call *Cijacking*, we came up with *CijScan*, a methodology for finding the traces of Cijacking. Our approach leverages the observation that a GitHub project's workflow on a the CI platform is well documented, either in a DevOps configuration file stored in its Github repository [75] or in the platform's log once the operations on the project complete. So CijScan automatically analyzes execution log on CI platform. Given this observation, CijScan is designed to automatically analyze both the configuration file and the log, looking for the information related to cryptomining, such

as wallet addresses and ~~names~~ domains of mining pools directly embedded in the command parameters or indirectly included in the file pointed to by the parameters, and the outputs of cryptomining ~~algorithms~~ jobs recorded in the log (Section III-A). Running ~~this approach over 540K~~ CijScan on over 580K Github repositories, we discovered 1,974 instances of Cijacking, which involve 12 different cryptocurrencies and almost all CPU-intensive mining algorithms. ~~This~~ Up to our knowledge, this is the first time that Cijacking instances have been ~~publicly disclosed, up to our knowledge~~discovered on a large scale.

~~Findings~~**Measurement and discoveries**. Looking into these Cijacking instances, we found that such cybercrimes appeared on CI platforms as early as March 2014 but the cases have started to surge since May 2017 with the price of Monero soaring, and continue to go up until now. During this period, we observed the evolution of attack strategies~~from the commitment logs of related GitHub repositories~~, which apparently ~~aims~~ aim at evading the protection put in place by CI platforms: for example, renaming cryptomining tools to avoid detection, terminating a mining ~~task~~ job before timeout by the platform to avoid getting blocked, etc. Also some cybercriminals continuously execute mining ~~tasks~~ jobs from their repositories, stopping right before the timeout each time and immediately launching a new one, for as long as 33 months on 4 CI platforms (which was still going on during our study, until at least July 2020). In the meantime, some miscreants are found to maintain multiple code repositories, as many as 297, for dispatching mining ~~tasks~~ jobs to multiple CI platforms simultaneously. These attacks are made possible by the generous free trial services offered by popular CI platforms, which allow one GitHub repository to utilize a large amount of resources (e.g., 8 GB memory) for 7 to 14 days, and then switch to a free plan with a lower level of resources allocated (e.g., 1 GB memory) indefinitely. We found that some cybercriminals open new GitHub accounts to sign up for the CI platforms for a new round of free trials after their current trials end. Our incomplete estimate shows that such Cijacking attacks has brought in at least 793,836 dollars in 2017 ($20,890 ~~/month)~~per month). We have reported our findings to both GitHub and affected CI platforms~~and are helping them address these new threats~~, which have acknowledged the presence of the abuses we discovered and removed them from the platform.

Mitigation. Defending against cryptojacking is known to be hard. Existing techniques are mostly based upon signature-based detection, looking for the patterns in mining code [73], [59], [47] or their runtime statistics (CPU, memory usage, etc.) [70], [49], [46]. These approaches are either susceptible to code obfuscation, or incurring false positives that mistakenly flag legitimate operations. To address these challenges, we developed a novel mitigation technique for protecting CI platform, which ~~leverages~~ uses the unique features of ~~crytomining~~ cryptomining to introduce runtime delays with asymmetric impacts on Cijacking and legitimate ~~tasks~~jobs. More specifically, ~~crytomining~~ cryptomining needs to accomplish a series of small tasks, each within a short time window. Failing to complete such a task results in loss of its related revenue and waste of all computation invested in the task. So our approach, called *Cijitter*, strategically injects a small delay to each time window to suppress a miner's revenue~~. These delays, however, only slightly reduce the throughput of a CI platform for serving legitimate users, as all computation performed on a legitimate task still counts towards its successful completion and the platform can re-allocate its resources to other tasks during the delays. Further our approach exploits the fundamental properties of hash computing mining algorithms entail to prioritize the delays to~~ , with only minor performance impacts on legitimate CI jobs. Cijitter leverages the observation that intensive hash computing at the center of every mining algorithm inevitably leads to high frequent memory-page access, and prioritizes the interference with the visits to ~~the pageswith very high access rates, which helps reduce the impact on legitimate tasks~~these pages. Unlike the signature-based detection, this strategy avoids delaying legitimate jobs when they do not have such memory-access behavior (as in most cases) and only moderately slows them down when they have, while significantly affecting the revenues of all mining tasks running under time constraints. Our theoretic analysis and experimental evaluation show that Cijitter ~~almost eliminates the revenue of Cijacking , while preserving 96% of the CI platform's throughput and ensuring 92~~is able to render Cijacking unprofitable, while incurring less than 10% overheads for 94.3% of legitimate ~~tasks to run to completion, with the waiting time of them prolonged by no more than 10~~CI jobs and reducing the throughput of a CI platform by merely 4%.

Contributions. The ~~contributions of the paperare outlined as follows~~paper's contributions are outlined below:

• *New discoveries and new understanding*. We report ~~the *first*~~ a systematic study on cryptojacking of public CI platforms, which ~~for the first time,~~ unveils real-world Cijacking ~~attacks~~ instances and their impacts. Unlike in-browser cryptojacking, these new attacks are found to work well with the new updates of cryptomining algorithms and become increasingly aggressive, thereby significantly undermining the public CI services. Our finding brings to the spotlight this new threat and the challenges in addressing it.

• *New techniques*. We developed a novel solution to the Cijacking threat, through strategic injection of delays to the processing of individual projects. Our approach causes asymmetric impacts on legitimate and illicit mining ~~tasks~~jobs, eliminating the revenues expected by the cybercriminals (so as to disincentivize them from exploiting an CI platform) with small impacts on legitimate ~~tasks~~jobs. Our analysis demonstrates that the gain the platform can achieve by using our protection significantly outweighs the price it pays.

~~**Roadmap**. The rest of the paper is organized as follows: Section II provides the background of our research; Section III describes our methodology for finding Cijacking; Section IV~~

reports our findings through a large-scale measurement study on the new attacks; Section V elaborates the design, analysis and evaluation of our mitigation techniques; Section ?? discusses the limitation of our study and envisions the future research; Section VI surveys the related prior research and Section VII concludes the paper.

We released the datasets and the source code of our techniques online [23].

II. Background

A. Cryptocurrency Mining

Cryptomining process. Cryptomining is a process to validate transactions on a blockchain network, and a miner who accomplishes the assigned task is awarded with new cryptocurrency. More specifically, the miner collects transaction data from the blockchain, validates it, and inserts it into the blockchain as a block. Once the block has been successfully added, the miner gets revenue (the cryptocurrency) from the blockchain. During this process, the miner has to solve a mathematical puzzle for transaction validation, which requires the miner to perform a significant amount of trial-and-error with a high frequency through a one-way hash function to find a nonce that produces the target hash value as Proof-of-Work (PoW).

To boost generation of PoWs and increase revenue, software and hardware optimizations based on GPU, application-specific integrated circuit (ASIC) ASIC and FPGA have been widely used for specific hash algorithms [60]. However, such technologies lead to a monopoly on cryptomining: a small set of participants with a huge amount of computation power can control a blockchain, which runs against its decentralized design principle [65], [47]. To prevent such accelerated computing from dominating the blockchain networkhardware, many cryptocurrencies [31], [68], [43] today have updated their algorithms to resist ASIC-based mining. Prominent examples include RandomX [31], CryptoNight [68], Argon2 [43], etc., which are designed to operate more efficiently on the commodity CPU with a large amount of memory [58] : e.g., execution of RandomX needs more than 2080 MB memory on x86 platforms.

to resist ASIC-based mining.

Cryptocurrency mining pools. As the difficulty of mining increases over time, a single miner can hardly gain enough block rewards to cover the cost for energy and computing resource spent. To improve their chance to find a block, miners today tend to work together, combining combine their computation resources to build mining pools, which can turn down their cost and improve their chance to find a block. Typically, a mining pool divides a mining task into the jobs job into the tasks with various difficulty levels and assigns them to miners. Each miner works on her job separately and if the job is completed within a time limit task separately and submits the result (a partial PoW) to the pool to have her workload certified. A validated result gives the miner a share of the reward should a block be found (based upon the difficulty of the job). Note that this pooled mining process is time-critical: if a job task cannot be finished successfully within a time limit (before a new block is found), its result becomes invalid and all effort invested will be in vain. For example, Monero has an average block time of 2 minutes, within which all jobs assigned by its mining pools need to complete.

Upon finishing a job task in time, a miner earns a "share" to keep track of her contribution to the progress of block discovery. Once a block is found, miners are rewarded according to the accepted shares they hold. Although the profit of the block can be distributed in various ways to ensure fairness, such as PPS, PPS+ [45], PPLNS [66], etc., those who have more shares certainly get more.

B. Container and DevOps Service

Docker container. Docker container is a runtime under OS-level virtualization, which can create isolated and standardized computing environments for an application. Such an application, its configuration files and library dependencies are packed into a static *image* of the container by running a series of commands documented by a *Dockerfile*. After instantiating a container from its image, its inclusion of dependencies provides a consistent environment for development, testing, and production of an application. Compared with other virtualization solutions such as virtual machines [69], container is characterized by less resource-consuming service support, fast startup, higher I/O throughput and lower latency. With these benefits, Docker container today has been widely adopted in the CI/CD pipeline to facilitate and simplify application developmentwith high flexibility and low overhead, serving as the default runtime for almost all commercial CI/CD platforms.

CI/CD service model and pipeline. DevOps is a set of practices that combine software development (Dev) and IT operations (Ops), aiming at accelerating software development through continuous integration and continuous delivery [61]. The pipeline of CI/CD includes building, testing and deployment stages, with each of them running as a task job and their ordered executions forming a workflowon today's CI/CD platforms, such as CircleCI [10], Travis-CI [34] etc. Such a workflow is set up by the developer who configures a Docker image for each taskjob, which allows her to build and test her application in the container. More specifically, the developer specifies a configuration file in her repository, which indicates the runtime and workflow to build or test her application on CI/CD platforms. For this purpose, the platforms often provide a large amount of computing resources to handle different kinds of tasks. The repositories using The repositories used for the CI/CD platforms are commonly hosted on GitHub, GitLab, and Bitbucketcommonly. All CI/CD platforms support that developers use accounts of GitHub, GitLab, and Bitbucket to easily login. However, the GitLab and Bitbucket are more facing to private developers and focus on protecting their privacy. Thus, we select GitHub as the target for our next study.

~~Use policy~~Terms of use, entry requirements and protection. As stated in their terms of service, all CI platforms prohibit any behavior harmful to their services, such as denial of service, malware spread~~, etc. Also~~ . In particular, most of them, such as ~~Travis-CI~~TravisCI [34], Buddy [8], Bitrise [7] and GitLab-CI [14], explicitly forbid any parties to use their service for cryptomining. If a user account violates such a policy, ~~he could be denied~~ it will be denied for access to the CI platforms and their services, with ~~his~~ the account suspended. However, one can open ~~an unlimited number of~~ many new accounts to use the services again and again. Actually, GitHub users and those associated with other repositories can conveniently log into these CI platforms using their repository credentials, so one can easily acquire free services from ~~the~~ CI platforms by opening new accounts on the repositories with different email addresses. ~~Although CI platforms often have some protection in place, such as keyword-based detection, against cryptomining, it can often be easily evaded (e.g, through obfuscating sensitive words). Further, the low entry bar for these platforms allows those getting caught to show up again and again with new accounts.~~

C. Threat Model

~~We consider the adversary intent on~~ We consider an adversary who aims at utilizing free CI services to stealthily run ~~mining tasks~~ cryptomining jobs for profit. For this purpose, the adversary opens accounts on free software development repositories like GitHub, GitLab and Bitbucket and crafts a platform-specific configuration file for each of his project to specify the workflow involving cryptomining ~~tasks~~ jobs to be executed on the CI platforms associated with these repositories. We assume that the adversary is capable of creating a large number of repository accounts, each with a very small ~~but non-zero cost. This models what happens in the real world: indeed one can easily open many GitHub accounts with a large number of real-world email addresses; acquisition of these emails, however, involve some~~ cost (e.g., passing the CAPTCHA test on Google).

We ~~consider the CI platforms that~~ also observed that mainstream CI platforms commit a large amount of resources (CPU cycles, large memory~~an~~ , and stable network connections) to support free trials and free plans~~, as they are doing today~~. The procedure to execute the workflow specified by a project's configuration file is public, which starts with known launch commands. Also these platforms ~~are known to~~ will monitor the progress of each CI task, recording its runtime statistics like usage of CPU, memory, disk, network and I/O~~, etc. Further we assume that they have already put some protection in place against illicit cyptomining, such as performance-based or keywords-based detection, so a straightforward attack cannot succeed.~~

.

III. Cryptojacking on DevOps Platform

A. Cijacking Discovery

~~Compared with personal devices, CI platforms like TravisCI, CircleCI, Azure-pipeline are characterized by the availability of a large amount of free computing resources, public access, loose protection, which make them a more ideal breeding ground for cryptomining. Indeed, there are anecdotal reports about illicit mining programs masquerading as legitimate CI tasks on these platforms [26]. However, up to our knowledge, no attack instances has been disclosed, not to mention presence of any in-depth analysis on the threat in the public domain. So in our research, we started off with an attempt to find such illicit tasks~~ Here we elaborate on the design and implementation of *CijScan*, our methodology for finding Cijacking instances on popular CI platforms~~, for the purpose of determining whether they are indeed posing a realistic threat and if so how they operate~~.

Idea and architecture. Since all ~~tasks~~ jobs to be executed by a CI platform are defined by a project's ~~platform-specific~~ configuration file in its code repository, a cryptomining ~~task~~job, if exists, must be triggered by the commands in the file. Further running a mining ~~task~~ job requires a *wallet address* for receiving rewards and ~~domain names~~ domains of *mining pools* when miners use these services (which is the most likely situation). As a result, these parameters ~~need to~~ will appear in the ~~configuration file~~ commands or in the CI platform's ~~log file that records the output (e.g., network communication)~~ logs that also record the intermediate meta-data produced when executing a ~~task~~job, should the ~~task indeed performs~~ job indeed perform illicit cryptomining. Based on the observations, we built *CijScan*, a simple scanner to ~~look for such parameters~~ find such parameters and intermediate meta-data in both commands ~~configured to be run by the platforms and log files.~~

and logs.

Fig. 1 shows the architecture of *CijScan*, including a *crawler*, ~~a~~ an *~~preprocessor~~interpreter* and a *mining detector*. The crawler is ~~built to retrieve~~ designed to collect CI-related ~~documents from GitHub(the configuration files) and CI platforms(log files). Then, the~~ repositories and their associated configuration files from GitHub, as well as their log files from CI platforms. From such a configuration file, the *~~preprocessor~~interpreter* ~~parses the commands in the platform-specific configuration file to identify their program name and parameters, and further analyzes the scripts (pointed to by some parameters) executed by the commands,~~ runs a *command parser* to extract commands and then a static *parameter identifier* to recognize their parameters. For parsing the parameters pointing to files, it further utilizes a *context constructor* to retrieve the filesystem of the related container image. Also, some commands are included in external scripts such as bash scripts ~~, Dockerfile , etc, to find additional program names~~ and Dockerfile (within the filesystem or hosted remotely). To recover these commands, the interpreter operates an *external script extractor* to collect the scripts and parse their commands and parameters. ~~It also parses~~ In addition, it analyzes the log files to ~~recover~~ find each command's execution ~~trace. From the program names, parameters and~~

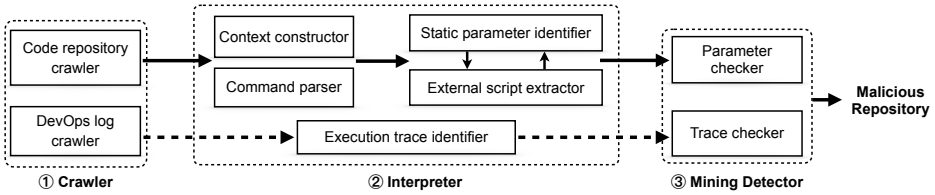


Fig. 1. CijScan Architecture.

traces discovered, the *mining detector* further seeks traces using an *execution trace identifier* (see interpreting below). These traces and command parameters are then used to search for wallet addresses and domain name names of mining pools . In addition to the necessary information for cryptomining, the detector looks for mining tasksby a *mining detector*. Other information to be sought from the execution traces is mining jobs' intermediate meta-data, such as including notification of hash rate, block detection, connection status of mining pools, from commands' execution history.

networks, feedback from mining process (e.g., share accepted, mining difficulty changed and new block detected). Report of any of these parameters (i.e., wallet address and mining pool, intermediate meta-data) leads to flagging of the associated repository as illicit (see mining detection below).

PreprocessingInterpreting. A project's On a configuration file, as stored in its GitHub repository, contains the commands for launching CI tasks. Execution of these commands is the only way for the file's corresponding CI platform to run these tasks. To analyze the commandswe bootstrap the *interpreter* by identifying its related container image. Such image carries information necessary for determining parameter values, including the content of a file as pointed to by a directory path. Specifically, to extract the image information, we developed a set of parsers to handle the unique configuration structures for different of different CI platforms, on top of *yaml* used by most of themthey use, according to their individual documentations. A challenge is that some commands can trigger external scripts, which run another set of commands. So the preprocessor also includes a *external script extractor* (Fig. 1) to download and parse the scripts. these platforms' documentation. Further, our approach downloads the image to fetch its filesystem as the context information to support the command's parameter interpretation.

Given those configuration files and external scripts, we parse After that, the interpreter extracts the commands from the configuration file and parses the command parameters for mining detection. Cijacking instance detection. A problem here is that some parameters are just variables variables (e.g., $var), pointers (e.g., file path), or even being obfuscated (e.g., string encoding, string slicing), whose values can only be determined when executing their commands in the bash shell, while some other parameters are pointers to files documenting the real values to be used by the commands. To recover the values of such parametersbe hard to determine. To solve this problem, we built a the *static parameter identifier* on top of the open-source bash interpreter [4] to run each command through interpretation. In this way, we can get the content of a variable right before its command is run and locate the file with its value when the parameter points to a file (through a directory path). Note that there are some external commands the bash interpreter cannot handle, such as scripts . In this caseinterpret each command without running it. More challenging here is that some commands can trigger external scripts to execute other commands. To handle this situation, the interpreter simply outputs such commandstogether with their input parameter values.

Note that running the bash interpreter should happen under the right environment setting, since otherwise the information necessary for determining parameter values, such as uses the *external script extractor* to parse the scripts and extract additional commands. Note that these external scripts are usually kept inside the repository or its container images. Thus, the directory path leading to the parameter file, may not exist. Such an environment setting is recorded in a container image indicated by the configuration file. So our preprocessor first instantiates a container based upon the image and then runs the interpreter inside the container to recover all the parameter values of the commands discovered. In the meantime, the preprocessor also parses the log file of the target CI platform, to recover extractor first attempts to search local the repository and the image's file directories to locate the scripts. If unsuccessful, it inspects the names of the external files associated with download commands (such as *wget*, *curl* and *git clone*), as output by the *parameter identifier* to find those included in the command parameters and download them for a further analysis by the identifier. Note that the bash interpreter cannot handle some commands in the languages like Python. In this case, it simply outputs the commands together with their parameters for the mining detection.

In addition, for the log files crawled from the targeted CI platforms, the *execution trace identifier* statically recovers the traces for execution of the commands on the platform by parsing the output component in the log file. Such information, including the parameters and the traces, also serves as inputs for *mining detection*.

Mining detection. On With the command parameters and execution traces, our mining detector searches for indicators of cryptomining, which include wallet addressesand , domains of mining pools, and mining job's intermediate meta-data. Common cryptocurrencies have wallet addresses comprised of alphanumeric characters with lengths ranging from 27 24 to 100, or in the form of email addresses. The domain of a mining pool always starts with *"stratum+tcp://"* and ends with a port number. Leveraging this observation, our detector runs a

~~commands~~ parameter checker that utilizes regular expressions to locate the wallet address and mining pool domains from command parameters. Also to be inspected are execution traces collected from CI platform logs. From these traces, our ~~history~~ trace checker looks for not only wallet addresses and domains, but also other statistics output by cryptomining tools at runtime, such as hash rates, block detection notifications, the connection status to ~~mining pools~~networks (e.g., mining pools), etc., using a set of ~~keywords~~signatures (Appendix Table **??**). Whenever any of such information has been discovered, our detector ~~reports a possible Cijacking case.~~ flags the associated repository as illicit.

B. Cijacking Analysis on CI Platforms

Data collection. In our research, we ran our crawler on GitHub, which utilizes Bigquery [5] and GitHub API [20] to search across 100M repositories for the configuration files related to different CI platforms using their unique names: e.g., *.travis.yml* for ~~Travis-CI~~TravisCI, *./circleci/config.yml* for CircleCI. Also there are a small number of platforms (2 out of 23) that keep configuration files of their connected projects on the platform side. So the crawler also takes a look at the commitment record reported on each project's GitHub page, which keeps track of the CI platforms involved in building the project. This allows our crawler to retrieve the configuration files from the corresponding platforms. Altogether, we gathered CI-related information from 582,438 GitHub repositories confirmed to use some platforms before the end of May, 2020.

Validation and findings. In our study, we used five servers (32 cores AMD Opteron 6276, 16G memory) with a total of 192 threads to run *CijScan* on 582,438 GitHub repositories. It took *CijScan* 17 days to finish all tasks including the command and trace analysis and mining detection. Among all the components of *CijScan*, the context constructor within interpreter performed the most time-consuming task (10 days), since the container image needs to be downloaded from its registry. It only took 12 hours for the mining detector to flag illicit repositories.

Scanning these repositories and ~~their platform logs~~ the logs on their related CI platforms, our approach ~~discovered~~ flagged 894 repositories (0.153%) ~~with possible mining traces.~~ as illicit. To validate the ~~findings, we manually inspected all these repositories to extract mining code (source code , binary executables and scripts) and submitted them to~~ results, we replayed CI workflows of all 894 repositories and manually recovered the binary code of each process in the CI jobs, and further leveraged VirusTotal [71] ~~, which confirmed that the vast majority of them (29~~ to identify the processes launched by cryptomining tools. Further, we traced these processes to find out whether they had wallet addresses necessary receiving mining revenue. This manual analysis took 3 cybersecurity professionals 9 days to accomplish. An instance was flagged when all of the annotators reached a consensus. Here inter-coder reliability we measured (among the annotators) using Cohen's kappa coefficient [12] is 0.947. The annotated dataset is released online [23]. In this way, we confirmed that 865 out of 894 ~~, with~~ repositories are indeed illicit miners, which gives a precision of 96.79%~~) are indeed illicit miners.~~. In Section III-D, we discuss the potential missing cases under this analysis.

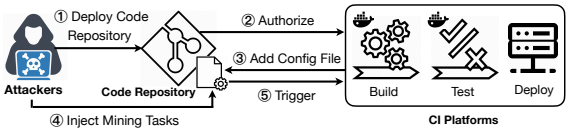


Fig. 2. Attack Overview.

```
1 FROM ubuntu:18.04 #Base Image
2 WORKDIR /
3 RUN apt install git make -y
4 RUN git clone <project> && make -j4
5 RUN wget <url:mining tool>
6 RUN ./<mining tool> -u <wallet id> -o <pool address>
7 ENTRYPOINT ./<project>
```

Listing 1. Malicious Dockerfile example. Line 2-4 represents a normal project building progress. Line 6 shows that mining command is inserted in RUN.

C. Cijacking Workflow

Attack process. Looking into the discovered Cijacking cases, we pieced together the attack's workflow as shown in Fig. 2. First the adversary needs to create a code repository on a public code-hosting platform (①), such as GitHub, GitLab, or Bitbucket, and authorize CI platforms to access the repository (②). Then he specifies a set of ~~tasks~~ jobs as the workflow to be run on each platform by creating a configuration file on the repository (③). Such a workflow includes normal, legitimate ~~tasks~~jobs, such as docker image building, source code compiling, and ones related to illicit cryptomining. In addition to injecting such a mining ~~task~~ job as a standalone ~~task~~ job in the workflow to the platform (④), the adversary could opt for a stealthier way to hide it behind a legitimate ~~task~~ job in one workflow. As an example, Listing 2 shows that mining commands are inserted into a job for docker image building, which are launched by command *docker build* and activated when the building progresses~~to the sixth line~~.

```
1 FROM ubuntu:18.04 #Base Image
2 WORKDIR /
3 RUN apt install git make -y
4 RUN git clone <project> && make -j4
5 RUN wget <url:mining tool>
6 RUN ./<mining tool> -u <wallet id> -o <pool address>
7 ENTRYPOINT ./<project>
```

Listing 2. Malicious Dockerfile example. Line 2-4 represents a normal project building progress. Line 5 demonstrates the mining tool is downloaded using bash command 'wget' while line 6 shows that mining command is inserted before an entrypoint.

~~Attack Overview.~~

Once the workflow is configured, the adversary needs to trigger it on a CI platform (⑤). This can be done automatically whenever the code in his repository is committed. Alternatively, he can start the workflow manually or through the CI platform's API, which enables him to launch a new cryptomining ~~task~~ job immediately after one finishes or times out.

TABLE I
THE TARGETED CRYPTOCURRENCIES AND THE MINING POOLS.

Type	Tool(s)	Pool(s)	% Cijacking instances
Monero	xmrig xmr-stak cpuminer	pool.minexmr.com supportxmr.com xmr.pool.minergate.com miningpoolhub.com nanopool.org pool.usxmrpool.com pool.monero.hashvault.pro xmr.crypto-pool.fr mine.ppxxmr.com xmr-eu.dwarfpool.com monero.lindon-pool.win pool.moriaxmr.com monerohash.com mine.xmrpool.net	67.67%
Darkcoin	cpuminer	drk.cpu-pool.net	24.51%
Bytecoin	xmr-stak minergate-cli	pool.bytecoin.party	3.95%
Litecoin	cpuminer	ltc.pool.minergate.com	1.87%
Cranepay	cpuminer	pool.cryply.io	0.80%
Electroneum	xmr-stak	uspool.electroneum.com pool.etn.spacepools.org etn.fairhash.org uspool.electroneum.com electroneum.hashvault.pro etn-us-east1.nanopool.org	0.60%
AEON	xmr-stak	mine.aeon-pool.com	0.20%
Fantomcoin	cpuminer cnrig	fcn.pool.minergate.com	0.15%
Arto (RTO)	xmr-stak	arto.cryptonight.me	0.05%
AIO	xmrig	aio.mine2.live	0.05%
Bitcoin	cpuminer	stratum.mining.eligius.st nicehash.com	0.05%
SHG	xmr-stak	pool.supportshg.com	0.05%
Sharkcoin	xmr-stak	coinshak.com	0.05%

Participants. As illustrated in Fig. 2, there are three parties involved in a Cijacking attack, as follows:

• *Attacker* is the adversary who abuses the computing resources of CI platforms to run cryptomining ~~tasks~~ jobs for profit. For this purpose, he needs to provide his wallet address to receive award and the domains of mining pools involved.

• *Code hosting platforms* provide code repositories and connect them to CI platforms. The repositories under the attacker's control include mining tools or scripts, and the configuration file for the CI platforms to specify how to launch a workflow with cryptomining ~~tasks~~ jobs on the platforms.

• *CI platforms* are the services the attacker abuses to run cryptomining ~~tasks~~jobs. Such a platform automatically processes a linked project from a code repository, committing resources to build a Docker image and test its code.

D. Discussion

Our study has made a first step towards understanding Cijacking on a large scale. However, we believe that our findings are limited by the vantage points taken in our study:

Private repositories and inactive accounts. Our experiment focuses on public repositories and active accounts, since the code and logs of private repositories and inactive accounts are not available for analysis. Note that starting from Jan. 2019, GitHub offers free unlimited private repositories [21], which can be utilized by the adversary to deploy Cijacking code.

Potential evasive cases. Our static analysis-based approach might miss some evasive repositories, e.g., those obfuscating their traces and using stepping stone servers to connect to mining pools (Section II-A). To evaluate the coverage of our approach, we randomly sampled and manually validated 10K repositories from the 580K CI-related repositories we collected (see the validation approach in Section III-B). Among them, 14 (0.14%) repositories were found to involve illicit mining, which have been discovered by CijScan independently.

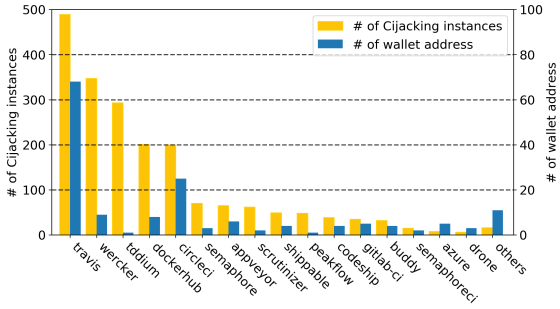


Fig. 3. Distribution of victimized CI platforms.

IV. MEASUREMENT

A. Landscape

Our study reveals that Cijacking is indeed trending in the CI platforms. Altogether, on 23 platforms, CijScan detected 1,974 Cijacking *instances* (~~aka~~ or simply a *Cijack*, ~~as identified by the combination of~~ which refers to a mining job launched by a unique GitHub repository ~~and one of its linked CI platforms~~on a CI platform) associated with 865 GitHub repositories, 607 Github accounts, 71 mining pools and 104 wallet addresses. Among all the instances discovered, 73.08% wallet addresses are used by at least 768 (38.9%) Cijacking instances. ~~Figure~~Fig. 3 further shows the distribution of Cijacking instances across different CI platforms. As we can see here, ~~Travis CI~~ TravisCI has the most Cijacks, followed by Wercker and Tddium.

Popular pools and cryptocurrencies in Cijacking. Looking into the wallet addresses and mining pool domains reported, we found 71 mining pools of 13 different kinds of cryptocurrencies involved in Cijacking. Interestingly, all discovered Cijacks turn out to use mining pools, as presented by Table I, which ranks these pools in the order of the number of instances they are associated with. From the table, we can see that miners tend to use the services of popular mining pools, those contributing relatively high hash rates to their corresponding blockchain networks: particularly, *xmr.pool.minergate.com* is the most popular pool for Monero (32.9%), followed by *pool.minexmr.com* (29.2%) and *pool.supportxmr.com* (22.4%).

Also our study shows that Cijacks have been used to mine almost all mainstream cryptocurrencies, from Monero, Darkcoin to Bytecoin and Litecoin. However, Monero is the most preferred one, being targeted by 1,336 (67.67%) Cijacks, which is possibly due to its highest mining profitability (based on the historical statistic of BitinfoCharts [6]).

Cijacking campaign discovery. We found that different Cijacks may share the same wallet addresses and/or Github accounts, indicating the presence of relations among their initiators (the attackers). To identify such relations across Cijacking instances, we built a graph for campaign discovery. In the graph, each Cijacking instance is represented as a node, and two instances sharing a wallet address or a Github account is described by an edge. All the instances on the

TABLE II
CIJACKING TOP-10 CAMPAIGNS IDENTIFIED BY WALLET ADDRESS.

# of campaign	# of CI platforms	# of repos	# of instances
Campaign - I	4	293	879
Campaign - II	11	107	296
Campaign - III	8	51	192
Campaign - IV	1	116	116
Campaign - V	5	14	70
Campaign - VI	1	46	46
Campaign - VII	2	16	32
Campaign - VIII	5	4	20
Campaign - IX	3	16	16
Campaign - X	2	16	16

connected graph formed in this way are considered to be in the same campaign. Altogether, 30 campaigns with ~~at least~~ 1886 instances ~~each~~ in total have been discovered. Table II presents the top-10 campaigns containing most instances, with the largest one including ~~892~~ 879 Cijacks. These campaigns describe a *lower bound* for the impact an attacker can have on the platforms, because one may have multiple wallets, which are not observable to us due to the limited information.

~~The targeted cryptocurrencies and the mining pools.~~

Looking into individual campaigns, we found that many of them enable attackers to concurrently execute a lot of Cijacking instances across multiple platforms: on average, an attacker exploits 4 platforms, with 15.7 instances running on each platform during the same period of time. Actually, such attackers do not even bother hiding relations among these tasks: on average, 3.39 instances from the same repositories, 1.07 from the same GitHub accounts, and 62.86 have the same wallet addresses are found from each campaign to be active during the same period of time. Particularly, the owner of the wallet address *"soku2.ko"* creates 248 GitHub accounts and names each account using a list of most popular given name (AlexaBierm, BarbaBecke, ChrisAchen, etc.). On the other hand, there are situations where the attackers apparently try to avoid sanctions imposed by CI platforms: we observed that in 17 campaigns, Cijacking operations start from some GitHub accounts right after they end in other accounts.

~~Fig. 4 presents the Jaccard similarities between the sets of campaigns on different platforms. We can see that the Cijacking campaigns on *Bitrise* and *Cirrus* platformsare most similar. Also~~Also, we observed some campaigns across multiple CI platforms. Particularly, 27 campaigns ~~are observed on the~~ were run on *CircleCI*, ~~*Travis-CI*~~*TravisCI*, *Wercker*, and *GitLab-CI* ~~platforms simultaneously.~~ simultaneously. We measured the relations between the sets of campaigns observed from two different CI platforms using Jaccard similarity coefficient, as illustrated by the grid between them (on x-axis and y-axis respectively) in Fig. 4. ~~Such co-occurrences happen when the two CI platforms have~~ The coefficient here demonstrates how campaigns are distributed across different platforms, especially those platforms with similar running environments ~~. Again, such a practice enables the attackers to~~ and operation modes. For example, the campaign set on *Bitrise* is highly similar to that on *Cirrus*, since their environments and settings are quite close. In this way, the attackers can maximize their profits ~~on different CI platforms~~ by re-using the same attack scripts on another platform.

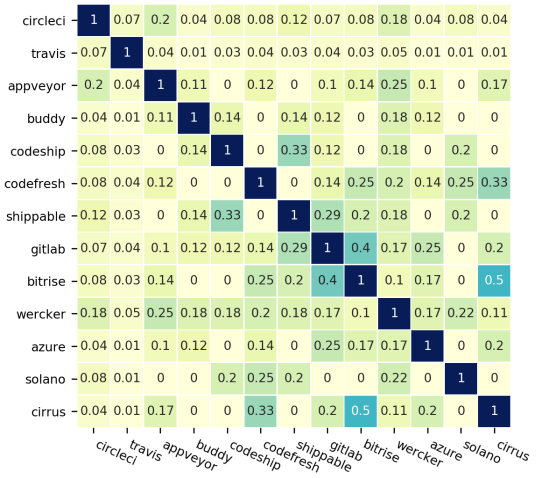


Fig. 4. Jaccard similarities on different platforms.

B. Cijacking Scripts

Mining tools. Most cryptomining tools allow for configurable parameters, which limits a miner's CPU usage ~~(i.e., throttle)~~ and thread number. We manually analyzed all 865 Github repositories related to Cijacking discovered and extracted their mining configurations (CPU throttle, number of threads, and running time). We found that 46.58% of mining-related scripts are present in Dockerfile while the rest are in the other files (e.g., Configuration file, Makefile). Also, most of Cijacking instances (72.34%), that is, the workflows containing mining code uploaded to individual platforms, set the average CPU throttle to 80% and the thread number to 4, in contrast to the browser-based ~~crytojacking~~ cryptojacking (typically 25% CPU throttle) [51]. Interestingly, from the commit logs of these GitHub repositories, we observed that Cijacking miners updated their scripts to change the parameter settings to balance between gaining enough profit and avoiding detection from CI platforms.

Time-out mechanisms. CI platforms tend to utilize time-out mechanisms to defend against resource abuse: a platform (e.g., ~~Travis-CI~~TravisCI, Wercker, CircleCI) terminates an instance running more than 4 hours (time-limit timeout), or without producing any output via *stderr* or *stdout* (i.e., no-output time-out). Also, CI platforms block the instances frequently triggering time-out. We found that Cijacking miners responded to such protection with evasive tricks to avoid being blocked. For example, most miners utilize the *"timeout"* command to end their Cijacking tasks within 3 hours. An interesting exception is "*fewa342rwr@tutanota.com*", whose workflow establishes an SSH reverse channel with the attacker and terminates before time-out, apparently, by acting upon remote instructions. This remote control does not leave much trace, as compared to running the "time-out" command, since the termination is not recorded by the platform-side log. Also, to avoid the no-output timeout, the workflow of "*fewa342rwr@tutanota.com*" generates random strings as output.

Instance triggering method. As mentioned earlier, Cijacking miners frequently re-launch Cijacking instances to avoid time-

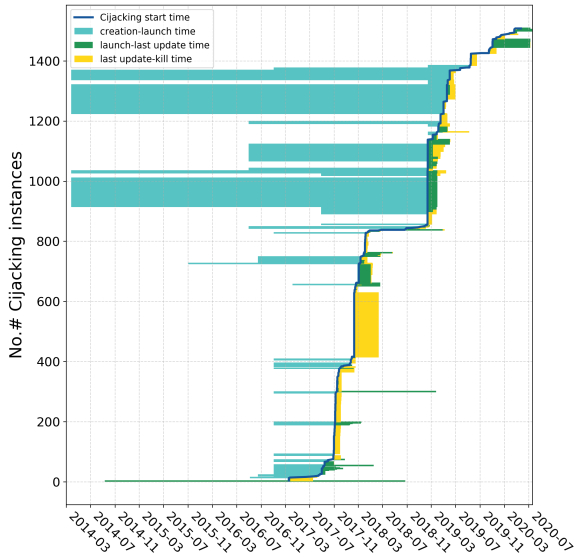


Fig. 5. Lifecycle of Cijacking instances.

out (which can lead to blockage). Particularly, from the logs from CI platforms, we found that each Cijacking instance has been re-launched 105 times on average.

To launch an instance on a CI platform, one could commit an update to the code repository, or trigger an instance manually or through APIs (e.g., *POST /workflow/{id}/rerun* [11]). In our study, we observed that a Cijacking miner "*jerolamo.r*" committed 46,496 updates on 16 code repositories over 4 months (02/17/2018 to 06/19/2018) to run 48 Cijacking instances on 3 CI platforms.

Code obfuscation. CI platforms (e.g., CircleCI) scan configuration files for the names of mining tools. To ~~avoid~~ evade the detection, Cijacking miners usually rename the tools to meaningless words or after common web services (e.g., Nginx, Node). In the meantime, we found that CI platforms do not flag the keywords related to mining pools' connection information, such as their domain names, which CijScan utilizes for finding mining instances (Section III).

C. Longitudinal Study of Cijacking

Attack lifecycle. Different from the browser-based Cryptojacking [55], [59], [57], Cijacks leave their traces in the commit logs of Github repositories and instance logs of CI platforms, which allows us to profile their lifecycle. Specifically, in our research, we analyzed different timestamps from commit logs and instance logs, including the creation time of a Cijacking instance's repository (t_c), the first launch time of the instance (t_l), the last update time of its configuration file (t_u) and the ending time of the instance's last execution (t_k). To this end, we crawled the Github commit logs from the 865 repositories and the logs of their 1,467 Cijacking instances on 11 CI platforms from Mar 2014 to July 2020, using the first commit date of each instance's repository as its t_c and the last commit of its configuration file as t_u. Also to determine the instance's t_l and t_k, we tracked its earliest launching date and the last termination date.

Fig. 5 shows the lifecycle of each Cijacking instance, ranked by their launch times (t_l). The average lifecyle of a Cijacking instance (i.e., $max(t_k, t_u) - t_l$) spans 42.8 days, 2.46 times as long as the period during which the attacker uses its repository to control the instance (i.e., $t_u - t_l$). Also the average life time of these instances' repositories is 337.8 days (i.e., $t_u - t_c$). Interestingly, the duration of controlling an instance from a Cijacking repository ($t_u - t_l$), which is 17.4 days, is in line with the length of free-trials provided by many CI platforms (14 days on average).

We observed that from 2017 to 2019, the number of Cijacking instances increased rapidly. 86.18% of the instances found in our study appeared during this period. Meanwhile, 57.01% of the Github repositories associated with Cijacking instances were created between Jan 2014 and Jan 2018. Interestingly, there is a time gap between the creation of a repository t_c and the first launch of a Cijacking instance from the repository (the earliest t_l associated with the repository) for 34.23% of the instances and the average length of such gaps is 861.5 days. Looking into these instances, we were surprised to find that 100 of these repositories likely changed hands among different accounts, since those making the first commits are not the current repository owners. When manually checking the historical content of such repositories, we observed semantic differences (e.g., demo a webapp vs launch a cryptomining tool). We contacted a repository's original owner, who responded that he was unaware of the change.

Of particular interest is the observation that 17.91% of the Cijacking miners rapidly adjust their Cijack scripts after their attack instances are launched. When manually investigating these updates in the commit logs, we found that those updates mainly aim at improving the effectiveness of Cijacking operations (e.g., testing different parameters to optimize mining tools, see Section IV-B, or updating mining tools or mining pools), changing attack targets (e.g., targeting different cryptocurrency), or adding code components for evading detection (e.g., code obfuscation, see Section IV-B).

Correlation between attacks and Monero prices. To understand the evolution of Cijacks, we monitored newly-appeared Cijacking instances over time. ~~Figure~~Fig. 6 illustrates the number of new Cijacking instances emerging per month from 07/2017 to 05/2020 and its relations with the change of the Monero price, with the price information coming from *coinmarketcap.com* [16]. From the figure, we can see that a large number of instances appears between 07/2017 and 04/2018, with the price moving toward its peak and such growth starts to slow down with the price going down. Indeed, the changes of the instances track closely with the dynamics of the Monero price, with the Pearson correlation coefficient [30] which measures the similarity of two data distributions, being 0.83. New Cijacking instances only spring up when the Monero is rising in price. The largest peak with an increase of over 231 new instances is observed around 02/2018 with the Monero price also hitting its peak $351.43. After that, new Cijacking instances decline to around 2.13 per month with the Monero

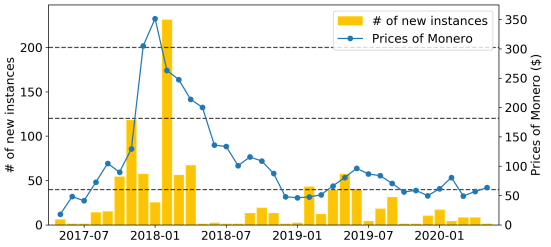


Fig. 6. Correlation ~~of~~ between Monero ~~rices~~ prices and ~~the number of new~~ cryptojacking ~~repositories~~trend.

price drops to $100.81.

Our hypothesis is that the depreciation of the cryptocurrency renders it less attractive to the new comers, who might opt for more profitable underground businesses.

D. Profit from Cijacking

~~Computing Power of Each CI Platforms per Adversary~~

To understand the economic incentives behind Cijacking, we estimate the revenue of Cijacking, and compare it with that generated by browser-based cryptojacking. Specifically, we utilize the model of browser-based cryptojacking [55] to identify the revenue per Cijacking instance, as follows~~:~~.

$$R_i = \frac{l_i \times h_i}{d} \times r_b$$

where R_i is the ~~total~~ revenue of a Cijacking instance i, l_i is the lifecycle (i.e., t_k-t_l, see Section IV-C) of ~~the instance~~ i, h_i is the instance's hash rate, d is the difficulty to mine a ~~block of cryptocurrency~~ cryptocurrency block, and r_b is the reward for each block discovered.

To estimate a Cijacking instance's hash rate h_i, we set up simulated CI environments locally for ~~all~~ 11 available CI platforms (based upon their hardware settings) and ran the mining algorithms on them. Specifically, we extracted the ~~task~~ runtime setting and the hardware environment of each CI platform using *procfs* [52]~~, as shown in Table ??~~. Then, we tested the underlying algorithms of the cryptomining tools collected, including CryptoNight and RandomX, with their default settings (4 threads), to measure their hash rates. (see hardware settings and hash rates in Appendix Table **??**).

To find out the mining difficulty d of the jobs issued by mining pools and the reward r_b for a new block discovered during a Cijacking lifecycle, we gathered the historical data about mining difficulties and reward per block from BitinfoCharts [6], a cryptocurrency statistics gathering platform.

With these parameters, we estimate the total revenue of the 1336 Monero Cijacking instances from March 2017 to May 2020 across 11 CI platforms to be $793,836.49 ($20,890.43 per month, when 1 XMR = US$117.62). Note that for the instances before Oct 2019, we use the hash rate of CryptoNight (the old mining algorithm of Monero), while applying the hash rate of RandomX (the new one) for other instances. This is because the algorithm of the Monero mining tools were ~~updated~~ migrated from CryptoNight to RandomX during that time.

Further, we ~~compare the~~ compared the monthly revenue of Cijacking with that of browser-based cryptojacking, ~~as estimated using~~ to find out how popular an illicit mining site needs to be for generating a comparable revenue as produced by a Cijacking campaign. For this purpose, we utilized the model and parameter settings of the prior work [55] ~~. We found that the monthly revenue of the aforementioned Cijacking instances is slightly above that of~~ to estimate the revenue of a cryptojacking site. Specifically, Campaign II discovered in our study (Table II) gained about $172,746.75 from March, 2017 to July, 2018 ($10,161.57 per month). This revenue is slightly higher than the browser-based ~~crypojacking on a Alexa top-500~~ cryptojacking on an Alexa top-1K website [1]~~. Note~~, which could generate 10,021.11 based upon its popularity, using the aforementioned model [55]. Also it is important to note that after the update of the PoW algorithm (Section II-A), which requires a large amount of memory, browser-based cryptojacking ~~can no longer mine Moneronow~~no longer works on Monero [13], [72].

V. Mitigation

A. Overview

Idea and design goals. Although CijScan reports 1,974 ~~real-world~~ Cijacking instances, which ~~has never been done before~~is the largest amount of real-world Cijacking instances being reported, detection of such illicit activities on CI platforms is challenging in general, particularly when the attackers are aware of the protection and make attempts to evade. More specifically, all the traces in configuration files can be obfuscated, with wallet addresses and mining pool domains being constructed only during a CI ~~workflow~~job's runtime. Further, the attacker could run a proxy to avoid ~~exposing such information, leaving no trace~~ leaving traces in the log file. Also, although machine-learning based cryptojacking detectors have been proposed by prior research, they are known to be fragile, easily affected by their runtime environments (e.g, presence of multi-users), and can be circumvented by a knowledgeable adversary [59]. So we need a more robust solution to mitigate the threat, without significant impacts on the performance of both CI platforms and the experience of their users.

A key observation in our research is that pooled mining operates under time constraints: a miner is expected to finish his job within a given time window or receive nothing in return for all the computation spent in the window. This fundamentally differentiates mining from a legitimate CI ~~task~~job, whose completion is not contingent upon the progress within each window but upon the accumulated effort across the windows. Leveraging this observation, we can come up with a strategy to periodically inject "jitters" into the progress of a ~~task~~job, in the hope to reduce the probability for each mining job to complete if it exists, but only moderately delay the progress of a legitimate ~~task~~ job and incur only a negligible impact on the CI platform's throughput~~, since it can always schedule the computing resources spared by the jitters to serve other tasks~~.

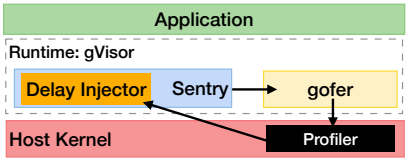


Fig. 7. Mitigation architecture

However, blind injection of jitters may not work well, which forces delay in the operations unlikely to be part of cryptomining. Alternatively, we can introduce jitters only at selected moments, when the activities triggered by a ~~task~~ job look less likely to be innocent: e.g., high-frequent visits to a certain page, indicating possible hash operations. Note that unlike detection, our approach is meant to slow down running of a suspicious ~~task~~ job a little bit, so we can focus on the ~~behaviors hard to avoid by~~ unavoidable behaviors for cryptomining, even though they may also be observed in many legitimate tasks. Following this line of thoughts we developed a new Cijacking mitigation approach, called *Cijitter*, aiming at two goals: (1) significant and robust impacts on cryptocurrency mining so the miner cannot profit, and (2) small impacts on CI services and moderate slowdown on legitimate tasks. Here we present the design, implementation and evaluation of the technique.

Architecture. Fig. 7 illustrates the architecture of our design, which includes two components: a delay injector to introduce jitters to a ~~task~~ job and a memory access pattern profiler to inspect the progress of the ~~task~~ job and determine when to cause delays. The key idea of our design is to monitor memory access for the sign of cryptomining, focusing on its *necessary condition* – a large amount of hash ~~computation that inevitably results~~ computations that inevitably result in intensive visits to certain code pages hosting the hash function. Once a set of pages have been found to have high access rates, jitters are injected into the corresponding process to slow it down, according to the time window of cryptomining.

In our research, we built this design on *gVisor*, Google's container runtime [39], with the profiler running inside a kernel module and the injector operating in the *Sentry* module under the user land, which interposes on the system calls from a specific process to manage its interactions with the kernel. Under the strict control of gVisor, Sentry is sandboxed and cannot directly communicate with the kernel. So our implementation utilizes the gVisor's gofer module [40] as a relay to establish a communication channel between the injector and the profiler. ~~Following we elaborate the design and implementation of the two components.~~

B. Design and Implementation

Cijitter monitors and interferes with the operations of every workflow on CI platforms. Whenever a ~~task~~ job within a workflow is launched, a profiler instance is initiated to track its memory access, identifying a set of pages that have been frequently visited. Such visits are then jittered with delays by the injector, as commanded by the profiler~~, through a Software Fault Isolation (SFI) mechanism~~.

Memory access pattern profiling. Our profiler is designed to monitor the memory access pattern of a process through manipulating the access bit of each page table entry (PTE), which is known to be much more efficient than instrumentation-based profiling [63]. Specifically, every PTE has a series of bits representing the status of its corresponding page frame including permissions, present, dirty, and access. When a page frame is accessed, its PTE's access bit is set. To evaluate the frequency of the accesses to a target page, the profiler periodically checks and clears its access bit.

With its relatively low overhead, access bit manipulation can still be too expensive if it has been used to profile every single page of a process, as the cost goes up linearly with the number of the pages. To address this problem, we utilize a sampling technique for efficient performance profiling [63], by dividing the ~~address space of the code segment~~ code address space into multiple regions and randomly picking up one page from each region to track. This strategy leverages the locality of reference [24], [44], a property that adjacent pages are more likely to have similar access patterns, which allows us to just sample one of them as a representative to understand the overall access pattern for the region.

Profiling the memory access patterns of a process is done through a sequence of inspection periods, with a *random* interval between two consecutive periods. During each period~~(around 100ms in our experiment)~~, the profiler repeatedly checks the access bit of each sampled page (one for each memory region), recording its status and clearing the bit if set. At the end of the period, it waits for a random interval before running the next set of inspections. The length of the interval is randomly drawn from a uniform distribution, whose mean is well below the size of the time window for a mining job, which ensures that each window has been adequately sampled. The access rate estimated from each inspection period is compared with that of the next one: a consistent high rate observed across both rounds indicates possible hash computation, which is characterized by persistent high-frequent memory accesses (Section II-A); On the other hand, if the observed access rate is irregular, the profiler may choose another page from the region to monitor.

A problem here is that repeated visits to a page may not be always observable from its PTE's access bit, since its virtual-physical address translation can be cached by Translation Lookaside Buffer (TLB), rendering its PTE bit unchanged during subsequent accesses until the page's TLB entry has been evicted. In practice, however, the size of today's mining algorithms tend to be large, covering hundreds of pages (e.g., RandomX with more than ~~2080~~ 2,080 MB), well above the size of TLB (typically 64 entries), so still a page's access rate can be roughly estimated from its PTE's status. Note that such a mining algorithm is carefully designed and cannot be easily changed without undermining its performance. Even when it is indeed changed to restrict memory visits to only a small set of pages, we can run TLB flush to evict the a page's TLB entry, to ensure the visibility of the access to the page at a moderate cost.

Delay injection. The outcome of the profiling instructs the injector to strategically slow down the accesses a process makes to a set of memory pages. Specifically, we analyzed the relation between the page access rate and the hash rate for leading cryptocurrencies through experiments, and then approximated such relations through linear regression. The linear functions obtained in this way (see Section V-C) are then utilized by the injector to determine a threshold for the access rate given a target hash rate. If the access rate to a sampled page goes above the threshold, the injector introduces a delay to each access to the page during the interval between inspection periods. The delay is determined by another linear function (see Section V-C) that maps an observed access rate and the threshold to the waiting time for each access.

Enforcing the delay to each access is done using the ~~SFI~~ Software-based Fault Isolation technique, through triggering a segmentation fault during each access to a page with permissions removed and then setting the permissions back during the fault handling after a pre-determined waiting time has passed. Specifically, the injector first makes the system call `mprotect` to clear all permissions of a target page including read, write, and execution, which induces a segmentation fault whenever the page is visited, causing a SIGSEGV signal to be issued. Upon receiving the signal, the fault handler in *Sentry*, which is instrumented by Cijitter, runs a sleep function to pause the current process for a duration given by the injector, as calculated from the delay function (`usleep`). After that, our instrumentation calls `mprotect` to restore the target page's permissions, allowing the access to proceed.

C. Security Analysis

To understand the security guarantee Cijitter can provide, we look into a profiling and delay cycle with a duration T, including an inspection period T_p and the inter-period interval T_d during which delays are injected into the target process. Since our approach does not rely on the differentiation between mining ~~tasks~~ jobs and legitimate ones, we consider any ~~task~~ job running on CI platforms potentially mining cryptocurrencies. So for each ~~task~~job, we derive its hash rate from the access rate observed through profiling to determine a delay for suppressing the revenue that can possibly be generated. For this purpose, we evaluated popular mining algorithms and utilized numeric analysis to derive two bijective functions: one maps an access rate to a hash rate and the other maps a hash rate to the expected mining revenue. Following we present an analysis that uses the functions to estimate Cijitter's impacts on mining revenue, operations of legitimate ~~tasks~~ jobs and the throughput of the CI platform, and further instantiate the estimates using an example for understanding the security implications of our approach.

Mining revenue. Let a be the access rate of the target page to be delayed, as observed by the profiler during the inspection period, and a_t be the new access rate for the page when the delays introduced by the injector kick in. We can estimate the upper-bound of the putative miner's expected revenue r_m (should all accesses serve the mining purpose), using two bijective functions derived through linear regression. Specifically, $F : a \to h$ is built for hash rate h estimate, based upon our testing of mining hash functions (RandomX), and $R : h \to r$ for expected revenue r estimate, through querying the profit calculator on the mining pool's website [27]. Following is our estimate:

$$r_m \leq R(\frac{F(a) \times T_p + F(a_t) \times T_d}{T}) \tag{1}$$

Note that for simplicity, we assume that the inspections performed by our profiler do not affect the access rate, so the access rate during the inspection period remains a. In reality, the rate should be lower, which brings down the hash rate and the revenue during this period. So the revenue above is just an upper bound.

Delay. To suppress the expected mining revenue, our approach forces the target process to sleep for a short period of time t_s to slow it down. For this purpose, our injector removes all permissions of the target page to induce a segmentation fault for any access attempt and then gives the permissions back after the delay, as mentioned earlier. This brings in a fixed cost, a delay t_f, for permission removal, context switch and fault handling. With these parameters, we can estimate the sleeping time t_s using function G, as follows:

$$G(a, a_t) = \begin{cases} \frac{1}{a_t} - \frac{1}{a} - t_f & (a > a_t) \\ 0 & (a \leq a_t) \end{cases} \tag{2}$$

So, given a and the target access rate a_t, our approach runs $G(a, a_t)$ to estimate t_s to be injected into each page visit. Note that although the delay takes its toll on the performance of the target process, its effect is limited when the original access rate a is low, as happens in many real-world legitimate ~~tasks~~jobs. Also the delay introduced by sleeping does not undermine the throughput of the CI platform, which can move the computing resources to other ~~tasks~~jobs.

Throughput loss. Further, we estimate the throughput loss introduced by Cijitter to the CI platform. For simplicity, we estimate the loss by looking at the total time wasted for controlling page access rates on the platform, including (pessimistically) the whole inspection period $T_p^{i,j}$ and the cost of the delay induced for each access $t_f^{i,j}$ during the follow-up delay period $T_d^{i,j}$, *across all profiling-delay cycles j of a ~~task~~ job and across all ~~tasks~~ jobs i*. This wasted time is compared with the total execution time across all ~~tasks~~jobs, which is lower-bounded by the accumulated duration of each profiling-delay period ($T^{i,j}$) across all the periods of each ~~task~~job, and across all ~~tasks~~jobs. Note that we do not count in the sleeping time $t_s^{i,j}$, since it can be recycled by the platform to serve other ~~tasks~~jobs. As a result, we have the throughput loss rate l estimated as follows:

$$l \leq \frac{\sum_i \sum_j (a_t \times T_d^{i,j} \times t_f^{i,j}) + \sum_i \sum_j T_p^{i,j}}{\sum_i \sum_j T^{i,j}} \tag{3}$$

~~It is easy to see that this rate approximates the ratio of the tasks that cannot be completed due to the overhead introduced~~

~~in a unit time, given an average execution time of such a task.~~

Analysis. Here we use an example to analyze the security guarantee offered by Cijitter, using parameters collected from real-world settings and experiments. Our experiments were performed on Intel ~~(R) Core(TM)~~ i7-4770 CPU ~~@ 3.40GHz~~ and 8G Memory. On the system, we first estimated F and G. Specifically, for F, we tested RandomX at different hash rates (from 50 H/s to ~~1500~~ 1,500 H/s) ~~for 10 times~~ and utilized linear regression to model the results: $F(a) = 0.36 \times a$ with the R-square 0.986. Further we used the data from ~~*pool.*~~*supportxmr.com* to model the relation between hash rates and expected revenues (XMR) per day: ~~$R(h) = 8 \times 10^{-7}h$~~ $R(h) = 8 \times 10^{-7}h$, with the R-square of line regression being 0.997.

The hash rate achievable on our system is 975 H/s, with a page access rate a of ~~2710~~ 2,710 times/s for the target page selected by Cijitter. Also the target hash rate h' set in our experiment is 40 H/s to ensure that the mining will not be profitable. Specifically, at this target hash rate, the revenue of mining, as estimated by G, is 3.2×10^{-5} XMR per day. During the free trials of CI platforms (mostly 14 days), the upper bound of one Cijacking instance's expected revenue becomes 4.48×10^{-4} XMR~~coins~~, about $0.01 (with the average price $41.6/XMR during 01/2020 - 05/2020). However, this income level does not make an attacker profitable, due to the cost for automatically registering GitHub accounts for continuing the mining effort after the free trials. In our research, we found that the cheapest service selling email accounts in bulk (for opening GitHub accounts) is $0.01 per email [18]. Also, the lowest price for solving GitHub CAPTCHAs is $0.0024 per account [9]. Therefore, the total cost per account $0.0124 is a little above the revenue ~~from~~ in 14 days $0.01, rendering the Cijacking unprofitable.

~~In the meantime, with~~ With the above target hash rate h', we estimated its access rate through F^{-1} to be 111 times/s. ~~Then we~~ We then randomly selected a 30-minutes window from the log file of ~~Travis CI~~ TravisCI to collect 386 ~~tasks~~jobs, and used their individual access rates ~~,~~ (as measured by our profiler~~,~~) and G to determine the delays to be injected to each ~~task~~job. In the experiment, we set the average length of a profiling-delay cycle to ~~5000~~ 5,000 ms and the inspection duration T_p to 100 ms. Also ~~we measured the~~ the each segmentation handling cost t_f ~~to be~~ about 0.4 ms ~~each time~~as we measured. Across all these 386 ~~tasks~~jobs, 67.1% of their ~~execution~~ elapsed wall time was delayed by our profiler. With all the parameters measured from the experiment, Formula 3 shows that the total throughput loss ratio caused by Cijitter is below 4.92%. In Section V-D we further report our evaluation of the delays introduced to individual ~~tasks~~jobs.

D. Evaluation

Setting. We evaluated the performance of our prototype of Cijitter and its ~~impact~~ impacts on the execution of legitimate ~~tasks~~jobs. All our experiments were conducted on Ubuntu 18.04, with the Linux kernel 4.4.0~~. Also we utilized gcc 5.4.0 to build the modified gVisor with libc-2.23.so as the LibC~~ library. As mentioned in Section V-C, , and the hardware settings ~~of our experiment~~ include a system with ~~an 4-cores~~ a 4 cores Intel i7-4770 CPU ~~with 8G Memory~~and 8GB memory.

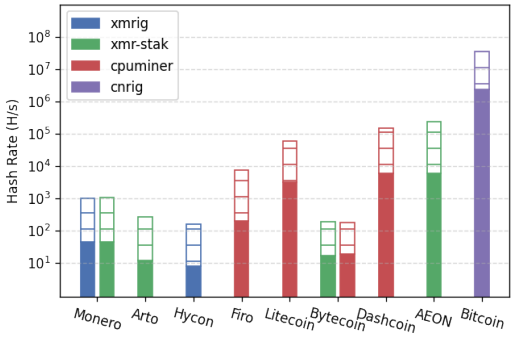


Fig. 8. Effectiveness on defending against different miners.

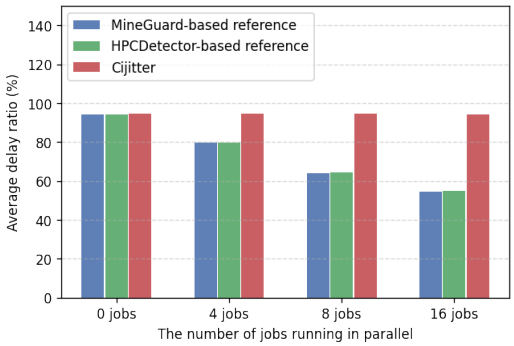


Fig. 9. Effectiveness of Cijitter and references.

~~Performance~~ Effectiveness of ~~micro-benchmarks~~Cijitter and comparison with state-of-the-art defenses. We first evaluated the ~~performance impacts of our approach using nBench [41], with the target access rate set to 111 per second, a rate that renders mining unprofitable (Section V-C). In our experiment, we ran each test case with and without Cijitter~~ effectiveness of Cijitter against all available mining tools on different cryptocurrencies. Specifically, we evaluated the effectiveness of Cijitter on *seven* cryptocurrencies found in CI platforms: Monero (algorithm: RandomX), Dashcoin (X11), Bytecoin (Cryptonight), Litecoin (Scrypt), AEON (K12), Arto (CryptonightArto), Bitcoin (sha256d), as well as *two* cryptocurrencies profitable to mine with CPU from the market [25], i.e., Hycon (Cryptonight V7) and Firo (MTP). Note that other cryptocurrencies found on CI platforms including Cranepay, Electroneum, Fantomcoin, AIO, SHG, and Sharkcoin are either inactive or can no longer be mined [15], [19], [17]. Hence, we ignore them in our experiment.

Fig. 8 shows the hash rate for mining different cryptocurrencies with and without Cijitter, and the blank part in each bar shows the hash rate declined by Cijitter (a.k.a., *delay ratio*). The result shows that the hash rate was reduced by Cijitter by over 95.3% on average. Among them, the lowest delay ratio is 93.1% on Bitcoin and the highest one is 97.3% on AEON. In addition, for Monero, Arto, Hycon and Firo, Cijitter can downgrade their hash rates to turn mining on these cryptocurrencies from profitable to unprofitable. Particularly,

mining Monero (the most profitable coin we studied) on CI platforms sees its profit in 14 days decrease from $11.70 to $0. Also, the 14 days' profit of Arto, Hycon and Firo went down from $1.04, $0.14 and $0.11, respectively, to $0.

We then evaluated the effectiveness of Cijitter in multitasking environment and compared it with the state-of-the-art detection methods MineGuard [70] and Conti et al. [46] (a.k.a., HPCDetector). To compare MineGuard and HPCDetector, we constructed two references, each using a different approach (based upon hardware performance counter, HPC) to replace the profiler for detecting suspicious jobs (MineGuard and HPCDetector, respectively) and then injecting jitters to these jobs as Cijitter does. We further ran these references and measured their effectiveness in decreasing hash rates on nine mining jobs, each using a different algorithm (i.e., RandomX, X11, Cryptonight, Scrypt, K12, CryptonightArto, sha256d, Cryptonight V7, MTP). Particularly, in our experiment (with 4-core Intel i7-4770 CPU and 8GB Memory), we ran a set of 0, ~~each for 10 times.~~

~~Table ??~~ 4, 8 or 16 concurrent benign CI jobs side by side with a mining job to simulate the multitasking environments of real-world CI platforms, as we observed from three platforms, including CircleCI, Wercker, and Codefresh (see Appendix Table VI).

Fig. 9 shows the average ~~execution time (Iterations/sec) under the two settings. Under Cijitter , the overheads of most benchmarks (in terms of delay) are all below 2%, with two exceptions: NUMERIC SORT (5.66%) and STRING SORT (2.6%)~~delay ratio on these mining jobs. The results show that Cijitter outperforms the references operating MineGuard and HPCDetector. Specifically, as shown in Fig. 9, the delay ratio introduced by Cijitter is always around 94.8% in all concurrent environments, while the ratios for the references go down quickly in the presence of multiple tasks, from 94.2% with a single job to 79.8%, 64.1%, 54.7% for the MineGuard-based reference, and 79.6%, 64.6%, 55.1% for the HPCDetector-based reference, when 4, ~~indicating that these applications do not perform~~ 8 and 16 jobs run concurrently. This is because both MineGuard and HPCDetector use the performance counter for job profiling, which is known to be non-deterministic under the multitasking environment [48], [74] and tend to introduce false negatives. By comparison, our profiler is designed to capture the jobs with elevated memory accesses, so it ensures a high coverage at the cost of false positives, which however causes little harm to a legitimate job except a small delay (see below).

Performance of benchmarks. We evaluated the performance impacts of our approach using PARSEC benchmarks suit [29] (with PARSEC3.0 and SPLASH-2) and SPEC 2017 benchmark [32]. Cijitter's overhead on PARSEC and SPEC are presented in Fig. 10 and Fig. 11, respectively. The results show that Cijitter introduced low performance overhead. More specifically, running Cijitter on all 27 PARSEC benchmarks (12 benchmarks from SPLASH-2), we observed the overhead (delay) on 21 of them below

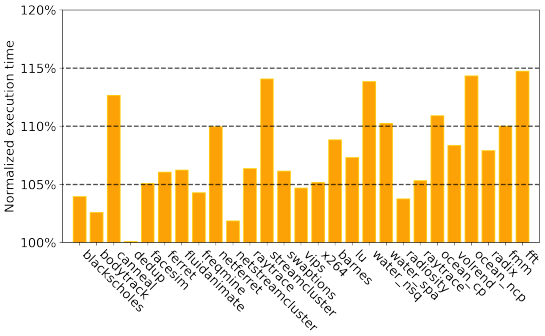


Fig. 10. Performance overhead on PARSEC.

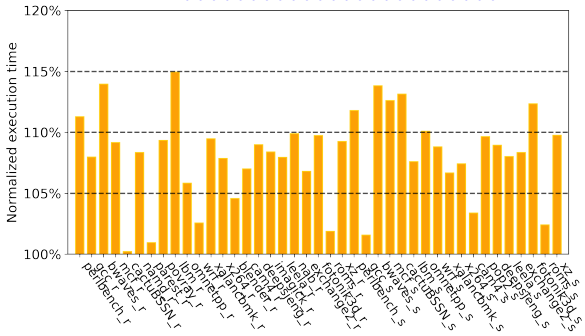


Fig. 11. Performance overhead on SPEC 2017.

10%. For the remaining benchmarks 'ocean_cp', 'canneal', 'water_nsq', 'streamcluster', 'ocean_ncp' and 'fft', our approach introduced an overhead between 10% and 15% (i.e., 10.91%, 12.66%, 13.85%, 14.60%, 14.32% and 14.73%, respectively), since they are all characterized by relatively intensive page accesses~~and~~ . For the SPEC 2017 benchmarks, the overheads of Cijitter are all lower than 15%, with 33 of the 41 benchmarks below 10%, indicating that our approach incurs only ~~negligible delaysto them~~small delays. The rest benchmarks, 'perlbench_r', 'perlbench_s', 'fotonik3d_r', 'mcf_s', 'actuBSSN_s', 'bwaves_r', 'bwaves_s' and 'lbm_r', were delayed by 11.28%, 11.78%, 12.33%, 12.61%, 13.13%, 13.81%, 13.96%, and 14.95% respectively.

Since Python is the most popular programming language used ~~by~~ in CI services like ~~Travis-CI~~TravisCI [50], we further evaluated our prototype on python benchmarks - pyperformance [38]. As shown in Fig. 12, 2/3 of the 15 benchmarks tested in our experiment suffer less than 8% delay. Besides, almost all benchmarks in pyperformance performed well on ~~our modified runtime~~Cijitter, with an overhead of 7.28% on average (geometric mean). Nevertheless, the execution of

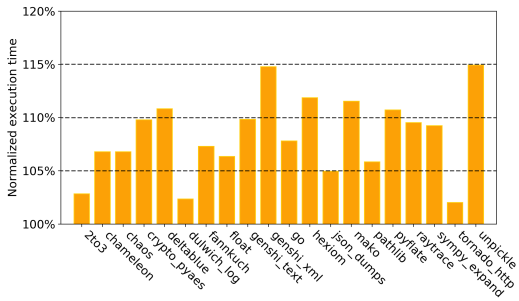


Fig. 12. Performance overhead on Pyperformance.

'genshi_xml' and 'unpickle' were delayed by 15% due to their frequent memory accesses.

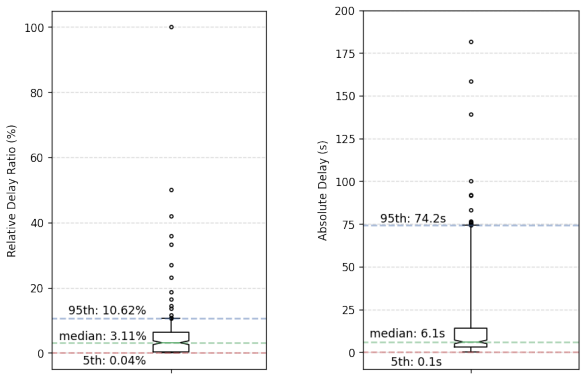
Performance of real-world ~~workflows~~CI jobs. ~~We chose 100 most frequently updated projects from Travis-CI, where each workflow of these projects includes 3 tasks on average. We re-built those workflows~~ To evaluate the performance impacts of Cijitter on real-world CI jobs, we randomly sampled 264 CI jobs (from the datasets of CI-utilizing GitHub repositories) and measured the ~~differences between their baseline performance (without Cijitter) and that observed under our Cijitter runtime environment~~elapsed wall time of individual CI job, when running them with and without Cijitter in place. Our evaluation shows that Cijitter introduces low impacts on the performance of CI jobs and developer productivity.

Fig. ~~?? presents the distribution of these workflows' tasks over the slow-downs measured in our experiment. As we can see here, our approach has only minor impacts (less than 10%) on the execution times of over 92%tasks. Even for the tasks such as packing/unpacking ('gradle assemble')~~13 presents the distributions of relative overheads (the percentage of performance degradation) and absolute overheads on CI jobs under Cijitter. The lower and upper bounds in the box plots are set to the 5th and ~~dependency management ('yarn install')~~95th percentile across all these jobs in terms of their overheads, and dots outside the boxes are outliers. As shown in Fig. 13(a), the median of these jobs' relative overheads is just 3.1%, with the 95th percentile at 10.6%. Overall, the overhead of 94.3% of CI jobs is below 10%. The outliers include the job with the maximum relative overhead of 63.5%, which however was only 4.2 seconds. The slowdown of all outliers is below 30 seconds (Appendix Table **??**). Regarding the absolute overhead (Fig. 13(b)), the median overhead is 6.1 seconds and its 95th percentile is 74.2 seconds with a relative overhead of 8.1%. Moreover, the jobs with absolute overheads above the 95th percentile all have lower than 9.8% relative overheads (Appendix Table **??**). Particularly, even though the maximum delay observed reaches 181.5 seconds, the job was only slowed down by 6.8%.

An observation is that some legitimate processes also calculate hash values, particularly those of package managers for installing new packages (e.g., *npm install*, ~~which are found to receive significant delays (>25~~*yarn install*), which need to perform heavy hash calculation for checking integrity of large packages (a behavior similar to that of miners). To avoid unnecessarily slowing down these processes, our approach automatically identifies them based upon the observation that they are typically invoked by a few common, legitimate package managers (such as *npm* [28] and *yarn* [37]), and utilizes a whitelist and checksum to identify their binaries so as to filter out related processes in our experiments.

Overall, our evaluation shows that Cijitter in general introduces low overheads to individual CI jobs and thus low impacts on developer productivity. According to prior studies [53], [42], the elapsed wall time of a CI job generally comes with measurable variations/delays (median: 8.5 minutes, mean: 19.64 minutes), as introduced by CI platforms, and a variation less than one minute is considered to have a very low impact on developer productivity. We also consider a job is slightly affected if the relative overhead is less than 10%. In our experiments, no CI jobs under Cijitter suffer from an overhead beyond this low impact level (i.e., with both an absolute overhead of over one minute and a relative overhead of at least 10%)~~due to their operations on large objects, they all successfully ran to completion within 50 minutes, the time-out limit set by Travis-CI.~~ .

~~Performance overhead on CI workflow.~~



(a) Relative overhead. (b) Absolute overhead.

Fig. 13. Performance overhead on CI jobs.

E. Discussion

A challenging issue is to understand whether the adversary can evade this protection, which requires reforming the mining algorithms to cause the checking of the memory access pattern to fail (Section V-B). In order to evade our defense, cryptomining jobs cannot continuously calculate hash values without being identified and delayed, since the memory access patterns of such operations are conspicuous and will be captured by our approach. However, any obfuscation attempts to avoid the patterns will inevitably bring down the hash rate, which impacts the revenue of the mining. Particularly, the random profiling techniques we utilize make it hard for the adversary to predict when the profiling stage starts so as to change the mining behavior accordingly. As a result, the hash operations have to be slowed down significantly to keep the mining under the radar, at the cost of revenue loss.

VI. Related Work

Illicit cryptocurrency mining has been studied for long. Huang et al. [56] studies Bitcoin mining ~~malwares~~ malware and revealed the operation of Bitcoin mining botnets on PCs. Pastrana et al. [64] conducted a large-scale measurement of cryptomining malware samples to analyze the underlying infrastructure. Konoth et al. [59] performed an empirical study on browser-based cryptomining ~~(i.e., cryptojacking)~~ and proposed a defense mechanism, along with a series of crypto-jacking studies [67], [55], [57]. Also, [22], [35] reported a set of insider (e.g., IT admins, unethical employees) attacks on cloud platforms to launch cryptomining tasks. ~~Different from~~

previous works, we report the first study on cryptomining of public CI platforms and unveil real-world Cijacking attacks and their impacts.

In terms of cryptomining detection, Tahir et al. [70] proposed MineGuardin addition to MineGuard [70] and HPCDector [46] discussed in Section V-D, SEISMIC [73] utilizes the unique features of WASM to detect cryptomining behavior on the cloud platforms . It employed hardware-assisted profiling to create discernible signatures for various mining algorithms. Conti et al. [46] detected core cryptomining algorithm using Hardware Performance Counters (HPC) to create clean signatures that grasp the execution pattern of these algorithms on a processorin browsers, which however cannot be moved onto cloud-based CI platforms due to the absence of the features on the platforms. Konoth et al. [59] proposed a browser-based cryptojacking detector MineSweeper, which is based on the identification of cryptographic functions through static analysis and monitoring of cache events during run time. Wang et al. [73] proposed a The static analysis of Minesweeper [59] is also designed for the browser architecture, and its dynamic analysis leverages HPC to find browser-based cryptojacking detector SEISMIC, which automatically instruments untrusted WASM binaries in-flight with self-profiling code to dynamically detect mining computations. However, these cryptojacking detection methods cannot be easily scaled to CI platforms . In contracts to these works, we developed a novel solution to mitigate Cijacking threat, through strategic injection of delays to the processing of individual projects.

cryptomining, which does not suit CI platforms (Section V-D). Darabian et al. [47] use the features extracted from the library call-chains as discovered in the Windows binary to detect Windows cryptomining malware; the static nature of the approach makes it vulnerable to an obfuscation attack, and also its precision-oriented design could reduce the coverage of cryptomining detection, a problem that Cijitter is designed to address.

VII. Conclusion

In this paper, we report the first a study on cryptomining of CI platforms. Unlike in-browser cryptojacking, this new type of attacks hide their mining tasks jobs behind the intensive computing for processing CI workflows, as legitimate tasks jobs also involve, and therefore become harder to detect. Our research has brought to light such new attacksand for the first time, unveiled the operations of such attacks, their pervasiveness, evolving strategies, lifecycle and revenuesrevenue. Further we present a novel mitigation technique to address the challenge of detecting those attacks. Our approach leverages the unique feature of cryptomining to strategically add delays so as to disproportionally affect mining jobs, rendering the miners unprofitable with only moderate overheads introduced to the CI platforms and most legitimate workflows. Both our legitimate jobs. Our discoveries and new technique have made an important step towards step toward better understanding the new trend of cryptojacking, contributing to more effective defense against the threat.

References

[1] Alexa. https://www.alexa.com/.

[2] Atlassian thwarts bitcoin mining attack on kubernetes environment. https://www.itnews.com.au/news/atlassian-thwarts-bitcoin-mining-attack-on-kubernetes-environment-523071/.

[3] Azure-pipeline. https://azure.microsoft.com/en-us/services/devops/pipelines/.

[4] Bash. https://www.gnu.org/software/bash/manual/bash.pdf.

[5] BigQuery: Cloud Data Warehouse — Google Cloud. https://cloud.google.com/bigquery.

[6] Bitinfocharts. https://bitinfocharts.com/.

[7] Bitrise - Mobile Continuous Integration and Delivery. https://www.bitrise.io/.

[8] Buddy: The DevOps Automation Platform. https://buddy.works/.

[9] Captcha test services. http://www.ttshitu.com/price.html.

[10] CircleCI. https://circleci.com/.

[11] CircleCI API. https://circleci.com/docs/api/v2/#rerun-a-workflow.

[12] Cohen's kappa. https://en.wikipedia.org/wiki/Cohen%27s_kappa.

[13] Coinimp stop monero service in browser. https://www.coinimp.com/news/coinimp-will-no-longer-support-monero-xmr-coin-mining.

[14] Continuous Integration (CI/CD) - GitLab. https://docs.gitlab.com/ee/ci/.

[15] Cranepay (crp). https://cryptorival.com/calcs/cranepay/.

[16] Cryptocurrencies Market Capitalization. https://coinmarketcap.com/.

[17] Electroneum pool. https://electroneum.miningpoolhub.com/.

[18] Email address transactions. http://ipcbuy.com/newsinfo.asp?id=48.

[19] Fantomcoin (fcn). https://whattomine.com/coins/102-fcn-cryptonight/.

[20] GitHub API v3 — GitHub Developer Guide. https://developer.github.com/v3/.

[21] Github unlimited free private accounts. https://github.blog/2019-01-07-new-year-new-github/.

[22] How to Get Rich on Bitcoin, By a System Administrator Who's Secretly Growing Them On His School's Computers. https://www.vice.com/en_us/article/nzzz37/how-to-get-rich-on-bitcoin-by-a-system-administrator-who-s-secretly-growing-them-on-his-school-s-computers.

[23] Illicit repositories dataset and cijitter codes. https://sites.google.com/view/cijitter.

[24] Locality of reference. https://en.wikipedia.org/wiki/Locality_of_reference/.

[25] Mining guides. https://f2pool.io/mining/guides.

[26] Mining Report on CI platforms. https://www.therepl.net/episodes/29/.

[27] Monero Mining Pool. pool.minexmr.com.

[28] npm package. https://www.npmjs.com/package/package.

[29] Parsec. https://parsec.cs.princeton.edu/.

[30] Pearson correlation coefficient. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.

[31] Randomx. https://github.com/tevador/RandomX.

[32] Spec. http://www.spec.org/index.html.

[33] Threat alert: Massive cryptomining campaign abusing github, docker hub, travis ci & circle ci. https://blog.aquasec.com/container-security-alert-campaign-abusing-github-dockerhub-travis-ci-circle-ci/.

[34] Travis-CI. https://travis-ci.org/.

[35] Us government bans professor for mining bitcoin with a supercomputer. https://bitcoinmagazine.com/articles/government-bans-professor-mining-bitcoin-supercomputer-1402002877.

[36] Wercker. https://wercker.com/.

[37] yarn package. https://yarnpkg.com/.

[38] The Python Performance Benchmark Suite, 2017. https://pyperformance.readthedocs.io/.

[39] gVisor, 2019. https://gvisor.dev/docs/.

[40] gVisor Security Basics, 2019. https://gvisor.dev/blog/2019/11/18/gvisor-security-basics-part-1/.

[41] Linux/Unix nbench, 2019. https://www.math.utah.edu/~mayer/linux/bmark.html.

[42] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of travis ci with github. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 356–367. IEEE, 2017.

[43] Dan Boneh, Henry Corrigan-Gibbs, and Stuart Schechter. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 220–248. Springer, 2016.

[44] Allan Borodin, Sandy Irani, Prabhakar Raghavan, and Baruch Schieber. Competitive paging with locality of reference. *Journal of Computer and System Sciences*, 50(2):244–258, 1995.

[45] Panagiotis Chatzigiannis, Foteini Baldimtsi, Igor Griva, and Jiasun Li. Diversification across mining pools: Optimal mining strategies under pow. *arXiv preprint arXiv:1905.04624*, 2019.

[46] Mauro Conti, Ankit Gangwal, Gianluca Lain, and Samuele Giuliano Piazzetta. Detecting covert cryptomining using hpc. In *Proceedings of the 19th International Conference on Cryptology and Network Security (CANS)*, page 344. Springer, 2019.

[47] Hamid Darabian, Sajad Homayounoot, Ali Dehghantanha, Sattar Hashemi, Hadis Karimipour, Reza M Parizi, and Kim-Kwang Raymond Choo. Detecting cryptomining malware: a d eep learning approach for static and dynamic analysis. *Journal of Grid Computing*, pages 1–11, 2020.

[48] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. Sok: The challenges, pitfalls, and perils of using hardware performance counters for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 20–38. IEEE, 2019.

[49] Dragos Draghicescu, Alexandru Caranica, Alexandru Vulpe, and Octavian Fratu. Crypto-mining application fingerprinting method. In *2018 International Conference on Communications (COMM)*, pages 543–546. IEEE, 2018.

[50] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F Bissyandé, and Luís Cruz. An analysis of 35+ million jobs of travis ci. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 291–295. IEEE, 2019.

[51] 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.

[52] Xing Gao, Benjamin Steenkamer, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. A study on the security implications of information leakages in container clouds. *IEEE Transactions on Dependable and Secure Computing*, 2018.

[53] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. *Empirical Software Engineering*, 24(4):2102–2139, 2019.

[54] Runchao Han, Nikos Foutris, and Christos Kotselidis. Demystifying crypto-mining: Analysis and optimizations of memory-hard pow algorithms. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 22–33. IEEE, 2019.

[55] Geng Hong, Zhemin Yang, Sen Yang, Lei Zhang, Yuhong Nan, Zhibo Zhang, Min Yang, Yuan Zhang, Zhiyun Qian, and Hai-Xin 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*.

[56] Danny Yuxing Huang, Hitesh Dharmdasani, Sarah Meiklejohn, Vacha Dave, Chris Grier, Damon McCoy, Stefan Savage, Nicholas Weaver, Alex C Snoeren, and Kirill Levchenko. Botcoin: Monetizing stolen cycles. In *NDSS*. Citeseer, 2014.

[57] 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.

[58] Hyunjun Kim, Kyungho Kim, Hyeokdong Kwon, and Hwajeong Seo. Asic-resistant proof of work based on power analysis of low-end microcontrollers. *Mathematics*, 8(8):1343, 2020.

[59] 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, 2018.

[60] Kay Kurokawa. Forking for asic resistance: A monero case study. https://perma.cc/5JL6-RPPS.

[61] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Computing Surveys (CSUR)*, 52(6):1–35, 2019.

[62] Monero Logo and Nicolas van Saberhagen. Monero (cryptocurrency).

[63] SeongJae Park, Yunjae Lee, and Heon Y Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*, pages 1–7, 2019.

[64] Sergio Pastrana and Guillermo Suarez-Tangil. A first look at the crypto-mining malware ecosystem: A decade of unrestricted wealth. In *Proceedings of the Internet Measurement Conference*, pages 73–86, 2019.

[65] Marc Pilkington. Blockchain technology: principles and applications. In *Research handbook on digital transformations*. Edward Elgar Publishing, 2016.

[66] Rui Qin, Yong Yuan, and Fei-Yue Wang. Research on the selection strategies of blockchain mining pools. *IEEE Transactions on Computational Social Systems*, 5(3):748–757, 2018.

[67] Jan Rüth, Torsten Zimmermann, Konrad Wolsing, and Oliver Hohlfeld. Digging into browser-based crypto mining. In *Proceedings of the Internet Measurement Conference 2018*, pages 70–76, 2018.

[68] M Seigen, T Jameson, Neocortex Nieminen, and AM Juarez. Cryptonight hash function. In *CRYPTONOTE STANDARD 008*, 2013.

[69] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, 66(105-111):2, 2014.

[70] 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.

[71] Virus Total. Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*, 2012.

[72] Said Varlioglu, Bilal Gonen, Murat Ozer, and Mehmet Bastug. Is cryptojacking dead after coinhive shutdown? In *2020 3rd International Conference on Information and Computer Technologies (ICICT)*, pages 385–389. IEEE, 2020.

[73] 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.

[74] Vincent M Weaver and Sally A McKee. Can hardware performance counters be trusted? In *2008 IEEE International Symposium on Workload Characterization*, pages 141–150. IEEE, 2008.

[75] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. One size does not fit all: an empirical study of containerized continuous deployment workflows. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 295–306, 2018.