BigBing: Privacy-Preserving Cloud-Based Malware Classification Service

Yunus Kucuk Nikhil Patil Zhan Shu Guanhua Yan

Department of Computer Science
Binghamton University, State University of New York
{ykucuk1, npatil5, zshu1, ghyan}@binghamton.edu

***Abstract*—Although cloud-based malware defense services have made significant contributions to thwarting malware attacks, there have been privacy concern over using these services to analyze suspicious files which may contain user-sensitive data. We develop a new platform called BigBing (a big data approach to binary code genomics) to offer a *privacy-preserving* cloud-based malware classification service. BigBing relies on a community of contributors who would like to share their binary executables, and uses a novel blockchain-based scheme to ensure the privacy of possibly user-sensitive data contained within these files. To scale up malware defense services, BigBing trains user-specific classification models to detect malware attacks seen in their environments. We have implemented a prototype of BigBing, comprised of a big data cluster, a pool of servers for feature extraction, and a frontend gateway that facilitates the interaction between users and the BigBing backend. Using a real-world malware dataset, we evaluate both execution and classification performances of the service offered by BigBing. Our experimental results demonstrate that BigBing offers a useful privacy-preserving cloud-based malware classification service to fight against the ever-growing malware attacks.**

I. Introduction

Recent years have witnessed continued growth of malware species that harm the trustworthiness of cyberspace. According to G Data, a computer security firm based in Germany, it took only 4.2 seconds to observe a new malware specimen in the first quarter of 2017, reduced from 4.6 seconds in 2016 with a yearly total of 6.83 million [23]. Against such ever-growing malware threats, the existing signature-based approach to malware defense has been found increasingly useless in practice [24], [37]. To counter against the humongous malware threats in the cyberspace, a large body of research efforts have applied machine learning techniques to automate malware defense (e.g., [22], [39], [34], [31], [36]).

Machine learning-based malware defense works effectively only if there exist sufficient data and computational power to train accurate predictive models. For a large number of end users – either institutional or personal – who do not have access to such resources, their only plausible solution is to use cloud-based services that support machine learning-based malware defenses. Indeed, cloud-based malware defense services, such as VirusTotal [20] and now-discontinued Anubis [3], have made significant contributions to the community by offering free malware analysis services. These services use existing AV (Anti-Virus) scanners or customized malware analysis tools to analyze the binary samples uploaded by their end customers and then send back the malware analysis reports.

Unfortunately, cloud-based malware defense services provided to the end users may come at a hefty privacy cost. For example, it has been recently found that terabytes of corporate data, which contain a wide spectrum of sensitive data such as passwords, keys and proprietary trade secrets, have been leaked by an end-point anti-malware detection and response tool called *Carbon Black Cb Response* through its use of the free VirusTotal service [8]. On its *About* page, VirusTotal clearly states that "Files and URLs sent to VirusTotal will be shared with antivirus vendors and security companies so as to help them in improving their services and products. We do this because we believe it will eventually lead to a safer Internet and better end-user protection." [2]

Another challenge for cloud-based malware defenses is that it is hard to scale the services at the granularity of individual binary samples. With advanced polymorphic or metamorphic mechanisms [42], voluminous malware variants can be generated within a short period of time. Therefore, when a cloud-based malware defense service becomes popular, it becomes increasingly difficult for its operator to provision sufficient computational resources that can keep up with the pace of new malware attacks. Indeed, the free malware analysis service offered by Anubis has been discontinued, citing the reason as lack of "resources to maintain these tools and improve them to match an ever-changing malware landscape" [3].

Against this backdrop, we have been developing a new platform called BigBing (a **big** data approach to **bin**ary code **g**enomics), which offers a *privacy-preserving* cloud-based malware classification service. Like other cloud-based malware defense platforms, BigBing relies on a community of users to contribute their binary samples. Through BigBing, however, users' privacy is preserved because a user uploads *only part of each sample*, which alone cannot reveal any sensitive data. Using proper cryptographic primitives, BigBing receives a valid binary sample only if *multiple parties* have seen the same sample and uploaded complementary chunks that can be used to reconstruct the entire binary executable.

To facilitate scalable malware defenses, BigBing offers a classification modeling service: it trains *user-specific* classification models from datasets dynamically generated with distributions that closely match those seen in end users' environments. Instead of classifying individual binary samples for end users – which makes it hard to scale the service to process voluminous malware variants, BigBing returns the classification models to the users, who should use their *own*

978-1-5386-8442-9 /18/$31.00 ©2018 IEEE
DOI 10.1109/PAC.2018.00011

computational resources to apply these models for classifying binary executables seen in their environments. To accommodate various operational environments, BigBing allows a user to provide her specific operational constraints for training an actionable classification model. Moreover, while offering classification modeling as a service, BigBing respects a user's privacy by *not* requesting any information unless it is needed to train an appropriate classification model for her.

In a nutshell, our main contributions include:

- Inspired by a recent incident of user-sensitive data leakage from a cloud-based malware analysis platform, we have developed a blockchain-based method to support privacy-preserving sharing of binary executable files in a cloud environment;
- Different from existing cloud-based malware defense platforms that offer malware analysis at individual sample level, we have developed a system that offers malware classification as a service, which trains user-specific classification models to detect malware attacks seen in their environments. The service uses a PSI (Private Set Intersection)-based protocol for the server to infer the data distribution in a user's operational environment.
- Using big data computing platforms (e.g., Spark and Cassandra) and the Flask web application micro-framework, we have implemented a prototype of BigBing, comprised of a big data cluster, a pool of servers for feature extraction, and a frontend gateway that facilitates the interaction between users and the BigBing backend.
- Using a real-world Windows PE (Portable Executable) malware dataset, we have evaluated the malware classification service provided by BigBing from both perspectives of execution and classification performances.

The remainder of the paper is organized as follows. Section II introduces related works. Section III presents the architecture of BigBing as well as its privacy concerns. Section IV gives a blockchain-based scheme for binary sample sharing and Section V discusses how BigBing offers classification modeling as a service. Section VI provides the implementation details of BigBing. Section VII shows the evaluation results of BigBing. We draw concluding remarks in Section VIII.

II. Related works

Due to the voluminous malware variants, machine learning has become a popular method to automate malware defenses (e.g., [22], [39], [34], [31], [36]). The development of BigBing has been inspired by two practical challenges in this trend: how to preserve users' privacy in collecting malware data needed for training machine learning models and how to scale up services to a large number of users. Particularly, although there has been wide-spread privacy concern over data stored on public clouds [30], few works have addressed the privacy challenge of cloud-based malware defense services. Private set intersection was used in [43] to achieve malware confirmation in a privacy-preserving manner with focus on signature-based malware detection.

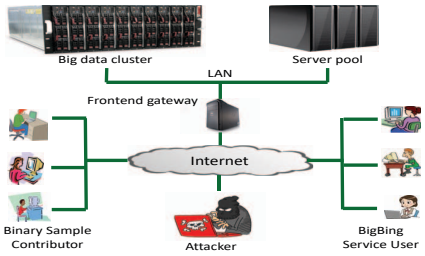


Fig. 1. The architecture of BigBing

Similar to BigBing, a few previous works [29], [33], [28] have used big data processing capabilities – particularly the Hadoop implementation of MapReduce – for malware clustering, triage, and similarity detection. A distinguishing feature of BigBing compared to these previous efforts is that it uses big data computing to train predictive models that are tailored to users' specific operational environments.

III. Architecture of BigBing

The design of BigBing has been inspired by the open cloud-based malware analysis platforms, such as VirusTotal [20] and now-discontinued Anubis [3]. It relies on a community of users (institutional or personal) to share their binary samples through its frontend gateway. BigBing stores these user-contributed binary samples on a big data cluster machine. BigBing uses a server pool to extract various features from both static and dynamic analysis of these binary samples. These feature data are also stored on the big data cluster machine.

To offer malware classification modeling service, BigBing requires an end user to upload a sample list of binary executable hashes. From this list, BigBing estimates the distribution of the binary executable data encountered in the user's operational environment, and then uses it to construct dynamically a training dataset that closely matches this distribution. Training a user-specific classification model helps improve classification accuracy when the model is deployed in her environment; for example, a university and a bank may need different binary executable classification models for detecting the same type of malware attacks.

As illustrated in Figure 1, the physical infrastructure of BigBing is comprised of three key components connected within a LAN (Local Area Network):

- *Big data cluster:* The big data cluster uses HDFS [5] to store a large corpus of binary executable programs as well as the feature data extracted from them. The big data cluster uses the Cassandra database [4] to manage these feature data. Moreover, Apache Spark and its machine learning library ML [6] are used to train classification models from the feature datasets.
- *Server pool:* The server pool includes commodity server machines to extract different types of features from binary executables. A server can run virtual machine-based sandboxes to observe their dynamic behaviors, execute a licensed IDA Pro application to disassemble them through

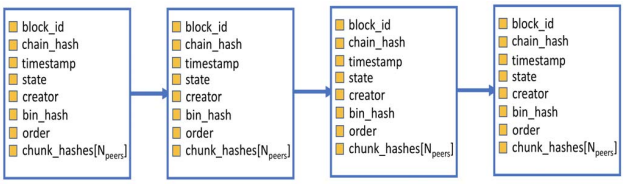


Fig. 2. Illustration of the blockchain for binary executable sharing

static analysis, or be simply used as a baremetal machine to run these samples in a sterile environment.

- *Frontend gateway:* The frontend gateway of BigBing offers a web interface to interact with both binary sample contributors and BigBing service users. A binary sample contributor can download a blockchain from the gateway, based on which she decides what binary samples she may be able to contribute to BigBing. On the other hand, through the gateway, a BigBing service user can request BigBing to train an actionable binary executable classification model that best suits her operational environment. To prevent malicious parties from abusing the service offered by BigBing, the frontend gateway enforces access control with password-based authentication.

Threat model. BigBing is designed to achieve two important property, *safety* and *liveness*, and this work considers any threats posed to each of these properties. Different from existing cloud-based malware analysis platforms, BigBing puts privacy preservation into its core design. *The safety property of BigBing concerns whether BigBing, if operated by the adversary, can obtain any sensitive data from its users who use or contribute to its service.* As BigBing relies upon a community of contributors to share their binary executable samples, a user may be concerned with uploading an executable program containing her sensitive data. As evidenced by the recent data leakage incident from shared malware [8], the user's concern is obviously *not* unfounded. Moreover, for BigBing to train a user-specific classification model, a user needs to upload a sample list of binary executable hashes. If BigBing itself does not have the binary executable program whose hash value appears on the list, it is totally unnecessary for BigBing to know it because it does not help improve the accuracy of the classification model trained but the hash value still reveals the user's sensitive information (i.e., the list of executable programs that the user runs in her environment).

The liveness property of BigBing concerns whether its operation can be disrupted by malicious users. The adversary may abuse the service of BigBing by uploading incorrect executable files or preventing legitimate users from uploading the correct binary samples. The liveness property of BigBing is to ensure that its operation should be resilient against this kind of DoS (denial-of-service) attacks by malicious users.

IV. Blockchain-based binary executable sharing

Our key idea of addressing users' privacy concern when sharing binary executable files with BigBing is to let a user upload only a chunk of each binary executable file, from which no sensitive data can be leaked to BigBing. Unless there are at least N_{peers} users, where N_{peers} is a system parameter, willing to upload distinct chunks, BigBing is unable to reassemble the original binary executable file and thus cannot obtain sensitive data about any of the users who have uploaded their portions. Our privacy model is sound: if multiple users have seen the same binary executable sample in their own operational environments, it is unlikely to contain data uniquely sensitive to any of them. In cases where an organization has multiple employees who may have received multiple copies of the same malware, they should share the same user account when contributing to BigBing.

To enforce this privacy model, we use a blockchain-based scheme to synchronize the users' knowledge about what portions of a binary executable sample have already been uploaded to the BigBing server. As BigBing operates on a cloud, it is natural for it to maintain the blockchain, alleviating the high overhead incurred due to a fully decentralized blockchain network such as Bitcoin [7] and Ethereum [10].

The blockchain data structure is illustrated in Figure 2. Each block contains the following fields:

- **block_id**: The genesis block has a *block_id* of 0, and every ensuing block increases its *block_id* by 1.
- **chain_hash:** The *chain_hash* field is used to check the integrity of the blockchain. It is calculated as the hash of the current block, except that its *chain_hash* field is filled with the *chain_hash* value of the previous block.
- **timestamp** (or *ts*): It gives the time when the current block is created.
- **state**: The *state* of a block can be `PENDING` (created by a client), `COMMITTED` (created by the server, indicating that a chunk of the binary executable file has been received by the server successfully), and `COMPLETED` (created by the server, indicating the the binary executable can be reassembled successfully).
- **creator:** The creator's ID is given by the field.
- **bin_hash:** It gives the hash of the binary executable file.
- **chunk_hashes**: For privacy preservation, a binary executable sample is transformed into chunks that satisfy the following requirement: a party cannot find any sensitive data from a binary executable file without obtaining all its chunks. We use *transform(b, h)* to define the transform of binary executable file b whose hash is h to N_{peers} chunks, and *reconstruct(chunks, h)* the reverse operation to reconstruct a binary sample b from its chunks obtained with *transform(b, h)*. We will discuss how to choose the *transform* and *reconstruct* functions shortly. To simplify verification, the *chunk_hashes* field of a block contains the hashes of all the chunks derived from the binary executable hashed to *bin_hash*, even though it may represent only one chunk of the binary executable sample.
- **order:** It gives the order number of the chunk, from 0 to $N_{peers} - 1$, represented by the current block.

Obviously the naive approach of partitioning the plaintext of a binary sample into N_{peers} pieces does not preserve privacy.

There are two alternative approaches:

- **SAE** (striping-after-encryption): Using a block cipher of 256 bits (e.g., Rijndael-256 [25]), the binary sample is encrypted with its own hash as the key. Chunk i, where $0 \leq i \leq N_{peers} - 1$, contains all the bytes in the encrypted file where their locations, j's, satisfy the following: $j \mod N_{peers} = i$. The SAE scheme works well in a practical setting where N_{peers} is small. Consider a practical setting where N_{peers} is 4 and the adversary wants to use a dictionary attack to recover the plaintext from partial ciphertext encrypted with the block cipher whose key is known. With k chunks in the ciphertext missing, the adversary needs to attempt $2^{256k/4}/2$ on average, which makes his attack effort equivalent to breaking a block cipher with a block size of $64k$ bits.
- **AONT** (all-or-nothing-transformation): We follow the all-or-nothing encryption algorithm described in [41]. The binary sample b is divided into $N_{peers}-1$ pieces, denoted as $\{b_i\}_{1\leq i\leq N_{peers}-1}$. Let $K_0(b)$ be the hash of the binary sample b stored in the *bin_hash* field in the block, and $K'(b)$ its hash with another hash function. Using $E_X(Y)$ to denote the encryption of Y with key X, the first i-th chunk, b'_i, where $1 \leq i \leq N_{peers} - 1$, is given by $b_i \oplus E_{K'(b)}(i)$ and the last chunk is $K'(b) \oplus h_1 \oplus ... \oplus h_{N_{peers}-1}$ where $h_j = E_{K_0(b)}(b'_j \oplus j)$ for $j = 1, 2, ..., N_{peers} - 1$. Due to all-or-nothing transformation, if all the chunks are available the reconstruction process can be efficiently done, but any missing chunk makes it hard to reveal any sensitive information contained in sample b [41].

When contributing binary executable samples to BigBing, each user executes a client-side protocol to interact with its server counterpart running on the cloud side.

Client: When a user whose ID is cid has a list of binary executables, denoted by $binlist$, to share with BigBing, she calls the CLIENT procedure, whose pseudocode is given in Algorithm 1. The user also maintains a local list, $checklist$, which stores the hashes of all the binary executables for each of which she has contributed a chunk to BigBing. Hence, if any of the binary executables on $binlist$ has its hash appear on $checklist$, it is immediately ignored to enforce the privacy policy of BigBing. The CLIENT algorithm uses a priority queue Q to process chunk uploading in an event-driven manner. Each element in the queue is the hash of a binary executable with its priority as the time it should be processed. Initially, the algorithm inserts the hashes of all the binary executables on $binlist$ into the queue, with their priorities all set to be the current time.

If the queue is not empty, at each step, the hash of a binary executable with the smallest time is scheduled to be processed at its corresponding time by calling procedure TIMER_HANDLER. Inside the handler, the client sends a request that contains the binary hash h and the size of its local blockchain to the server. In its response, the server sends back the remaining blockchain that the client has not seen yet. The client verifies that all the blocks must be valid, using their chain_hash fields (Line 17-22 in Algorithm 1), before it updates its local copy of the blockchain. The client then calls procedure UPLOAD to upload a chunk of the binary executable to the server (if necessary). Finally, the client extracts the next event with the smallest firing time from priority queue Q and schedules a timer for it.

When calling procedure UPLOAD, the client checks if the binary hash has already appeared on $checklist$ to ensure that no more than one chunk will be uploaded to the server. After passing the check, it performs the following to decide which chunk it should upload to the server: it goes through the blockchain and searches previous transactions on the same binary executable (based on its hash). If a chunk is in a state of `COMMITTED` (someone else has already uploaded the chunk) or `COMPLETE` (the server has reassembled the entire binary executable that matches the *bin_hash* field), it should not be re-sent. Otherwise, if a chunk is a state `PENDING` (someone else is trying to send the same chunk to the server, but it is not yet committed), the client checks if the transaction has expired (each transaction has an expiration period T_{exp}). If the transaction has not yet expired, the client does not attempt to upload this chunk for the moment. If the client can find a chunk that is *not* either committed, completed, or pending but not expired, it can proceed to upload this chunk. When doing so, it creates a new block with its fields properly filled and sends it to the server; meanwhile, it spawns a new child process to deal with uploading the chunk to the server and then adds the hash of the binary executable to $checklist$.

If all the chunks are pending and not expired, the client cannot find any chunk to upload at the current moment. In that circumstance, it inserts a new event to priority queue Q for the same binary executable, with its firing time scheduled at the earliest expiration time of all these chunks with a delay of $T_{exp}/2$ to absorb clock skewness.

Server: On the server side, it calls the SERVER procedure, which is given in Algorithm 2. The blockchain C is initialized with a genesis block, where its *block_id* field is 0, its *bin_hash* field is 0 (a non-existing binary executable), and its chain hash is calculated by assuming that the previous block's chain hash is 0. The main body of the SERVER procedure is a loop on processing each incoming request from a client. To prevent abuse of service, BigBing allows each user to send at most LIMIT requests for the same binary executable (Line 10 in Algorithm 2). Once it receives a legitimate request, the server sends back the remaining blockchain that the client has not seen and then waits for the client to send a new block until a timeout of τ time units expires.

The server verifies a new block sent from the client: its *chain_hash* field must be correct – given this block and the last block's *chain_hash* field, its *state* field must be `PENDING`, its creator must be the client that the server is talking to, its order number must be valid, its *block_id* must be that of the last block plus 1, and its timestamp is within T_{exp} time units relative to the current time. If the new block is valid, the server puts it at the end of its copy of the blockchain and then spawns a child process to accept a new chunk from the client. The

Algorithm 1 CLIENT(cid, $binlist$, $checklist$)

$Q \leftarrow$ an empty priority queue
for each binary executable e on $binlist$ **do**
  $h \leftarrow hash(e)$, $t \leftarrow get\_cur\_time()$
  if h is on $checklist$ **then** continue **end if**
  $insert\_with\_priority(Q, h, t)$
end for
if not $is\_empty(Q)$ **then**
  $(h, t) \leftarrow pull\_highest\_priority\_element(Q)$
  Schedule a timer to be fired at time t with parameters $(h, C, cid, binlist, checklist, Q)$
end if

procedure TIMER_HANDLER($h, C, cid, binlist, checklist, Q$)
`request:`
  Inform the server of the beginning of a transaction
  Send a request (h, $size(C)$) to the server for the blockchain
  Wait for response C' from the server
  $prev\_h \leftarrow$ last block's $chain\_hash$ in C (or NULL if $size(C) = 0$)
  for each block b on C' **do**
    $cur\_h \leftarrow b.chain\_hash$, $b.chain\_hash \leftarrow prev\_h$
    if $cur\_h \neq hash(b)$ **then go to** `request` **end if**
    $prev\_h \leftarrow cur\_h$, $b.chain\_hash \leftarrow cur\_h$
  end for
  $C \leftarrow C + C'$ ▷ Append C' to the tail of C
  Call UPLOAD($h, C, cid, binlist, checklist, Q$)
  Inform the server of the end of transaction
  if not $is\_empty(Q)$ **then**
    $(h, t) \leftarrow pull\_highest\_priority\_element(Q)$
    Schedule a timer to be fired at time t with parameters $(h, C, cid, binlist, checklist, Q)$
  end if
end procedure

procedure UPLOAD($h, C, cid, binlist, checklist, Q$)
  if h is on $checklist$ **then** return **end if**
  Initialize array exp of size N_{peers} with MAX_INT's
  $chunks \leftarrow transform(binlist[h], h)$
  for $i = 0$, ..., $npeers - 1$ **do**
    $chashes[i] \leftarrow hash(chunks[i])$
  end for
  for each block b on C **do**
    if ($b.bin\_hash \neq h$) **or** ($b.order \notin \{0, ..., N_{peers} - 1\}$) **or** ($b.chunk\_hashes \neq chashes$) **or** ($b.state$ = PENDING and $get\_cur\_time() - b.ts > T_{exp}$) **then**
      continue
    end if
    if $b.state$ = COMMITTED or COMPLETED **then**
      $exp[b.order] \leftarrow -1$
    else if $b.state$ = PENDING and $exp[b.order] > 0$ **then**
      $exp[b.order] \leftarrow \min(exp[b.order], b.ts + T_{exp})$
    end if
  end for
  $uploaded \leftarrow false$, $min\_exp \leftarrow$ MAX_INT
  for $i = 0$, ..., $npeers - 1$ **do**
    if $exp[i] =$ MAX_INT **then**
      $lb \leftarrow$ last block on chain C
      Create a new block nb with $nb.block\_id = lb.block\_id + 1$, $nb.chain\_hash = lb.chain\_hash$, $nb.state$ = PENDING, $nb.creator = cid$, $nb.timestamp = get\_cur\_time()$, $nb.chunk\_hashes = chashes$, $nb.bin\_hash = h$, $nb.order = i$
      $nb.chain\_hash \leftarrow hash(nb)$
      Send nb to the server, and also spawn a child process to transfer $chunks[i]$ to the server
      Add h onto $checklist$, $uploaded \leftarrow true$
      break
    else if $exp[i] > 0$ **then**
      $min\_exp \leftarrow \min(min\_exp, exp[i])$
    end if
  end for
  if $uploaded$ is false and $min\_exp <$ MAX_INT **then**
    $insert\_with\_priority(Q, h, min\_exp + T_{exp}/2)$
  end if
end procedure

Algorithm 2 SERVER(sid)

$C \leftarrow$ genesis block, $buf \leftarrow \emptyset$, $cmap \leftarrow \emptyset$, $sem \leftarrow$ Semaphore()
for each transaction from any client cid **do**
  Obtain its request (h, len) for the blockchain
  if $len > size(C)$ **then go to** `FINAL` **end if**
  $count \leftarrow 0$
  for each block b on C **do**
    if $b.bin\_hash = h$ and $b.creator = cid$ **then**
      $count \leftarrow count + 1$ **end if**
  end for
  if $count >$ LIMIT **then go to** `FINAL` **end if**
  Send all the blocks on C from len to the end to client cid
  Schedule a timer to be fired after τ time units
`WAIT:`
  Wait for the timer to fire or a new response r from client cid
  if response r is a new block from client cid **then**
    $cur\_h \leftarrow r.chain\_hash$, $lb \leftarrow$ last block on C
    $r.chain\_hash \leftarrow lb.chain\_hash$
    if $cur\_h \neq hash(r)$ or $r.state \neq$ PENDING or $r.creator \neq cid$ or $r.order \notin \{0, ..., N_{peers} - 1\}$ or $r.ts > get\_cur\_time() + T_{exp}/2$ or $r.ts < get\_cur\_time() - T_{exp}/2$ or $r.block\_id \neq lb.block\_id + 1$ **then**
      Cancel the timer, **go to** `FINAL`
    else
      $r.chain\_hash \leftarrow cur\_h$
      Append block r at the tail of C
      Spawn a child process calling RECV_CHUNK(r, cid) to receive a chunk of binary executable from client cid
    end if
    go to `WAIT`
  end if
`FINAL:`
  sem.Acquire(), move items in buf into $buf2$, sem.Release()
  if $buf2$ is empty **then** continue **end if**
  $lb \leftarrow$ last block on C
  for each $(h, o, chashes)$ in $buf2$ **do**
    Create a new block nb with $nb.block\_id = lb.block\_id + 1$, $nb.bin\_hash = h$, $nb.order = o$, $nb.chain\_hash = lb.chain\_hash$, $nb.state$ = COMMITTED, $nb.creator = sid$, $nb.ts = get\_cur\_time()$, $nb.chunk\_hashes = chashes$
    $nb.chain\_hash \leftarrow hash(nb)$
    Append nb at the tail of C
  end for
  $done\_set = \emptyset$
  for each $(h, o, chashes)$ in $buf2$ **do**
    if h in $done\_set$ **then** continue **end if**
    $ready \leftarrow true$
    for $i = 0, ..., N_{peers}$ **do**
      if $cmap[chashes[i]]$ is empty **then**
        $ready \leftarrow false$ **end if**
    end for
    if $ready$ is $false$ **then** continue **end if**
    $chunks \leftarrow \{cmap[chashes[i]]\}_{1 \leq i \leq N_{peers}}$
    $de \leftarrow reconstruct(chunks, h)$
    if $hash(de) = h$ **then**
      $lb \leftarrow$ last block on C
      Create a new block nb with $nb.block\_id = lb.block\_id + 1$, $nb.bin\_hash = h$, $nb.order = -1$, $nb.chain\_hash = lb.chain\_hash$, $nb.state$ = COMPLETED, $nb.creator = sid$, $nb.ts = get\_cur\_time()$, $nb.chunk\_hashes = chashes$
      $nb.chain\_hash \leftarrow hash(nb)$
      Append nb at the tail of C, add h to $done\_set$
    end if
  end for
end for

procedure RECV_CHUNK(b, cid)
  Wait for cid to transfer a chunk ch
  if $hash(ch) = b.chunk\_hashes[b.order]$ **then**
    sem.Acquire()
    Add ($b.bin\_hash$, $b.order$, $b.chunk\_hashes$) into buf
    $cmap[b.chunk\_hashes[b.order]] \leftarrow$ received chunk
    sem.Release()
  end if
end procedure

child process calls the RECV_CHUNK to receive the chunk from the client, and if its hash matches that of this chunk stored in the block, it saves the chunk inside a map, $cmap$, keyed by the chunk hash.

At the end of the main loop of the SERVER procedure, the server checks if any new chunks (not necessary from the client that the server just talked to) have been received. For each received chunk, the server creates a new block with its state set to be COMMITTED and appends it to the tail of the current blockchain with its *chain_hash* field properly calculated. It is also possible that a new chunk received allows the server to reassemble an entire binary executable. Using the chunk hashes as the keys to search $cmap$, the server reconstructs the binary sample. If the reconstructed sample has the same hash as the hash of the binary executable (i.e., h), it is valid and the server thus appends a new block with its state set to be COMPLETED and its *bin_hash* field to be h.

Protocol analysis. Our blockchain-based method for sharing binary executables satisfies the following properties:

- **Safety:** In this context, the safety property means that the server cannot obtain sensitive data from any of the contributors. The CLIENT procedure in Algorithm 1 uses its local $checklist$ to ensure that for any binary executable at most one chunk should be sent to the server, which achieves our privacy goal.
- **Liveness:** The liveness property here means that if there are at least N_{peers} users who have the same binary executable and they have sufficient bandwidth to upload a chunk of the executable within T_{exp} time units, the server will eventually be able to obtain this binary executable. Our method achieves liveness because if a contributor finds that she is not able to upload a chunk at the current moment (because all the chunks are in a PENDING state without expiring), she will retry after a certain period of time. For a non-cooperative user, he may choose *not* to upload the correct chunk after each request, which makes the corresponding block expire after T_{exp}. However, as each user can create at most LIMIT requests on behalf of the same binary executable (Line 10 in Algorithm 2), a cooperative user will eventually be able to replace him in uploading that chunk. It is possible that a malicious user can create multiple sybil accounts. But due to only a finite number of sybil accounts he can create, a *persistent* cooperative user will still be able to outlast these attempts and upload the correct chunk eventually. To mitigate the adverse effects of non-cooperative users, BigBing can further deploy anomaly detection methods, which are outside the scope of this work.

V. Malware classification modeling as a service

Through its frontend gateway, a user of BigBing's server needs to provide three types of inputs: *operational constraints*, *performance objective function and constraints*, and *a sample list of binary executable hashes*.

Operational constraints. A user's operational constraints concern what kind of features can be collected from her working environment. For example, although IDA Pro [14] offers comprehensive capabilities to disassemble executable programs, a user may not have a license for it to extract features. A cautious user may not want to use malware features collected from dynamic execution if she cannot find a malware execution sandbox with strong isolation.

Performance objective function and constraints. A user of BigBing's service also needs to input her performance objective that should be optimized. As there can be many different types of malware attacks against the user's network, BigBing uses multi-class performance metrics, instead of the traditional binary classification metrics. The formal definitions of the multi-class performance metrics supported by BigBing have been summarized in the Appendix. In addition to classification performance objectives, BigBing also allows the users to input their performance constraints. A Neyman-Pearson classifier, for example, maximizes the detection rate while keeping the false positive rate below a certain threshold [45]. Hence, for training a Neyman-Pearson classifier, the user needs to specify detection rate as her performance objective and an upper limit on false positive rate as her performance constraint.

Sample list of binary executable hashes. To estimate the data distribution in the user's environment, BigBing requires her to upload a sample list of binary executable hashes. The list is *uniformly* sampled from all her executable programs, benign or malicious. A higher sampling rate allows BigBing to recover the user's true data distribution more accurately, but at the price of a higher transmission overhead.

While BigBing requires its users to upload a sample list of binary executable hashes, a privacy-sensitive user may not want to reveal to BigBing all the executables that run in her environment. To alleviate users' privacy concerns, BigBing can engage a private set intersection protocol [27] with the user such that after the protocol, BigBing should only know the *intersection* between the executable program hashes on the user's sample list and the hashes of those executable programs stored on BigBing's big data cluster. There are efficient algorithms for private set intersection protocols [26], [40], and we will evaluate the execution performance of PSI operations used by BigBing in Section VII.

Optimization problem formulated from user inputs. The frontend gateway forwards the user's input to the BigBing backend, which trains an optimal predictive model that satisfies the user's requirements. Formally, let C_o and C_p denote the user's operational and performance constraints, respectively, and $G_p(f)$ be her performance objective. Also let the sample hash list be H, and its data distribution denoted by $\mathcal{D}_H$. Given a predictive model f, $G_p(f)_{|\mathcal{D}_H}$ gives a measure of the user's performance objective on the data generated from distribution $\mathcal{D}_H$, $C_o(f)$ the requirement that the features used by the predictive model f should satisfy C_o, and $C_p(f)_{|\mathcal{D}_H}$ the requirement that predictive model f should satisfy performance constraints C_p when applied on data generated from distribution $\mathcal{D}_H$. Hence, the user's expectation on the predictive model can be formulated as the following optimization problem:

$$\begin{aligned} \underset{f}{\text{Optimize}} \quad & G_p(f)_{|\mathcal{D}_H} \\ \text{Subject to} \quad & C_o(f) \\ & C_p(f)_{|\mathcal{D}_H} \end{aligned} \tag{1}$$

A. Training user-specific predictive models

Solving the optimization problem formulated as in Eq. (1) needs knowledge of distribution $\mathcal{D}_H$. The big data cluster of BigBing stores a large corpus of binary executable samples. It is a well-known challenge to find the ground-truth family labels for malware samples [32]. To find a labeled malware dataset, BigBing relies on the detection results of various AV scanners. We use the Zeus sample compiled from Github [21] as an example, whose MD5 hash is `fac741d0618b82cdf7c41c89cffdb19e`. The detection results of five major AV scanners are given in Table I. From their detection results, we extract the most unique keywords as the family labels, as in the last column of Table I.

TABLE I
DETECTION RESULTS BY DIFFERENT AV SCANNERS ON A ZEUS SAMPLE WITH MD5 HASH `fac741d0618b82cdf7c41c89cffdb19e`

AV scanner	Detection Result	Keyword
Microsoft	PWS:Win32/Zbot!ZA	Zbot
Kaspersky	Trojan-Spy.Win32.Zbot.bopd	Zbot
ESET-NOD32	Win32/Spy.Zbot.YW	Zbot
McAfee	PWS-Zbot.gen.ds	Zbot
Symantec	Infostealer	Infostealer

From the detection results by different AV scanners, BigBing currently applies the majority voting scheme [46] to label their corresponding families: among n AV scanners, if more than half of them identify it as a family label l, it is labeled as family l. However, the other types of heuristics can be used here to derive malware family labels, such as conflict-free malware labeling [44].

For each item on the user-provided sample hash list H, its label is decided through majority voting based on the detection results of different AV scanners (if the label cannot be decided, it is given as `unknown`). After processing all the sample hashes on H, BigBing calculates the frequency of each family label, and the derived frequency histogram is used to approximate the user's data distribution $\mathcal{D}_H$ in Eq. (1). For convenience, we let $\mathcal{D}_H(l)$ denote the frequency of family label l in distribution $\mathcal{D}_H$.

Let list B denote all labeled binary executables stored on the cluster. BigBing needs to sample list B to generate a training dataset whose distribution matches $\mathcal{D}_H$. To this end, BigBing examines each sample on list B to check if it satisfies the user-provided operational constraints, and if so, puts it onto list O which eventually contains all eligible samples that have been found to meet users' operational constraints. By abusing notation O slightly, we use $O(l)$ to denote the list of all samples in O with family label l.

BigBing next generates a training dataset T to train a predictive model that solves the optimization problem stated in Eq. (1). Taking into account its available computational resources, BigBing has an internal parameter m to control the size of the training dataset T. With inputs including distribution $\mathcal{D}_H$, the list of eligible samples O, and dataset size m, the sampling algorithm is given in Algorithm 3 to generate training dataset T. It is noted that a sampling scheme with replacement is used to generate dataset T. Moreover, for a family label that is present in $\mathcal{D}_H$ but not in O, it is ignored due to lack of representative samples available to BigBing.

Algorithm 3 Generation of training dataset T

Require: distribution $\mathcal{D}_H$, list O, parameter m
Ensure: Distribution of dataset T matches $\mathcal{D}_H$
  $T \leftarrow \emptyset$
  for each label l of interest in distribution $\mathcal{D}_H$ **do**
    if length of $O(l)$ is greater than 0 **then**
      $m_l \leftarrow m \cdot \mathcal{D}_H(l)$
      for $1 \le i \le m_l$ **do**
        $r \leftarrow$ a random executable program from $O(l)$
        Add r onto dataset T
      end for
    end if
  end for

Given the training dataset T, BigBing next searches for a predictive model that solves the optimization problem shown in Eq. (1). BigBing considers only those classification models F that have already been implemented on the Apache Spark platform, and for each hyperparameter used by classifier f in F, it keeps a list of plausible values. The hyperparameters of a machine learning model are the knobs that control the complexity or the capacity of the model but cannot be changed in model training. BigBing uses the grid search method to find the optimal classifier. For each classifier f in F, it considers all combinations of plausible hyperparameter values, under each of which BigBing uses the training dataset T to train an optimal classifier and then applies cross-validation to measure its average classification performances. If the parameterized classifier trained cannot meet any of the predefined performance constraints, it is rejected; otherwise, it is put into a candidate list L_f. After evaluating all possible combinations of hyperparameter settings for each classifier in F, the one among $\{L_f : f \in F\}$ that optimizes the objective performance function is chosen and returned to the user.

VI. IMPLEMENTATION

We have developed a prototype of BigBing, whose web interface is still under development. Its big data cluster is hosted on a Super Micro SuperServer (6028TR-HTR) with eight Intel Xeon E5 2620 8-core processors, four 128GB DDR-4 2133MHz ECC/REG RAMs, and four Seagate 4Tb hard disks. The big data cluster runs 64bit Ubuntu Linux of version 16.04, hosts an HDFS (Hadoop Distributed File System) [5], relies on Apache Spark and its ML library [6] for data processing and machine learning, and uses the Cassandra database [4] to manage malware samples and their features.

We use several workstations in our laboratory as the server pool of BigBing. Each workstation is equipped with an Intel i7-4790 8-core 3.60GHz CPU, 32G RAM, and a 2T hard disk. Each of these workstations runs 64bit Ubuntu Linux of version 16.04 and uses KVM/QEMU for virtual machine-based sandboxing. One of these workstations has a Linux-based license to run IDA Pro disassembler of version 6.9 [14].

The software component of BigBing consists of two parts: blockchain-based binary executable sharing and malware classification as a service. Based on the results from the simple Python LOC (Line of Code) counter provided at [1], the former part has 547 and 376 lines of Python code on the server and client side, respectively, and the latter is implemented with 7,122 lines of Python code.

The functionality of binary executable sharing given in Algorithms 1 and 2 has been developed on top of Flask, a Python micro-framework for web development [12]. We have implemented a web service which uses the Flask-based RESTful API for the communications between BigBing and any user who would like to contribute her binary executable samples. The client sends requests to the server for each transaction using the HTTP protocol, and all the requests and responses are transmitted in the JSON data format.

For Windows PE malware samples, BigBing uses the following tools to extract malware features (to differentiate the tools used for feature extraction, we follow the convention `X:Y` to denote feature type `Y` extracted by tool `X`):

- **pefile** [16]: pefile is a Python module to read and work with PE files. As shown in [46], both numerical and boolean features extracted from the PE headers are useful for predicting malware families [46]. We thus use `pefile` to extract malware features of type `pefile:numerical` and `pefile:boolean`.
- **hexdump** [13]: The Linux `hexdump` utility can be used to extract the frequencies of byte sequence n-grams as malware features of type `hexdump:n-gram`.
- **IDA Pro** [14]: We use a dedicated server running IDA Pro to extract the CFG (control flow graph) of each malware sample and extract structural information as malware features of type `idapro:cfg`.
- **Cuckoo** [9]: Cuckoo is a popular sandbox for malware analysis. We run PE malware samples within a virtual machine contained with KVM/QEMU. From the analysis report, we extract the number of times each API call has been made by the malware as its features of type `cuckoo:api`. For the network configuration inside the VM for malware analysis, we have four different modes: *Internet* (the VM has Internet access with offensive traffic blocked due to ethical concern), *simulated Internet with InetSim* [15], *simulated Internet with FakeNet* [11], and *no Internet simulator*.

The raw features extracted with the aforementioned tools are stored on the HDFS. We next preprocess the raw feature data into data formats that are readable by the Spark ML library. Using Spark, we perform imputation of missing features with their means, standard scaling, and feature selection on the raw feature data. The feature selection algorithm chosen affects the classification performance significantly [46]. As the feature selection algorithms implemented even in the latest Spark package perform poorly on malware data, we use the linear SVM model with a L_1-norm regularization term implemented by `scikit-learn` [19] to select at most 200 features for each feature type.

VII. Experimental evaluation

We use a Windows PE malware dataset with 15,983 unpacked and 6,480 packed samples belonging to 12 families, *Bagle, Bifrose, Hupigon, Koobface, Ldpinch, Lmir, Rbot, Sdbot, Swizzor, Vundo, Zbot,* and *Zlob*. Additionally, we use 494 benign PE programs. The distributions of both unpacked and packed malware samples are shown in Figure 3. In our experiments we use this dataset to evaluate the performance of BigBing from two perspectives: its *execution performance* and *classification performance*.

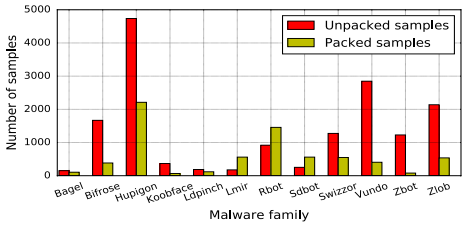


Fig. 3. Malware data distributions

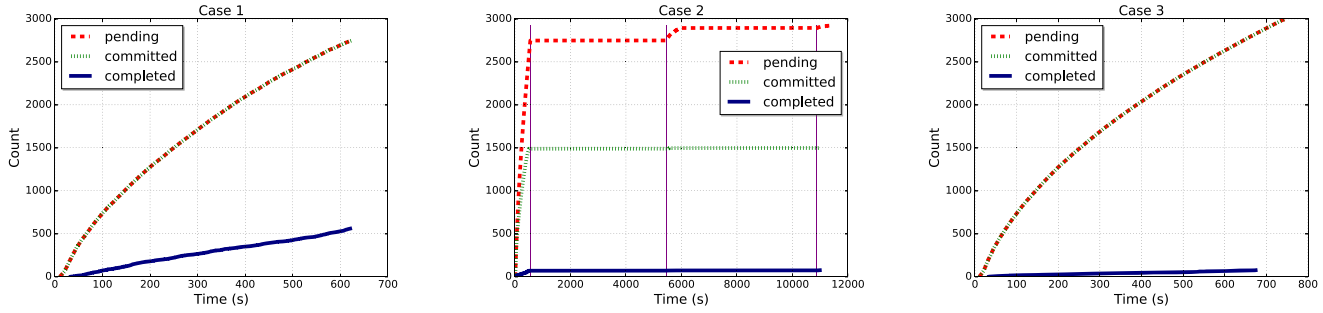
A. Execution performance of BigBing

In this part, we perform three sets of experiments to evaluate the efficiency of the BigBing platform.

(A.1) Blockchain-based binary executable sharing. In the experiments, we consider six clients, each of them possessing 500 malware executables randomly picked among 1317 Zbot malware samples. Each client establishes its first connection with the server after a random delay within 10 seconds. Afterwards, each client continually sends requests until all the samples on its list of binary samples have been contributed. The parameters in Algorithms 1 and 2 are configured as follows: $T_{exp} = 3600$ (sec), $N_{peers} = 3$, $LIMIT = 3$, and $\tau = 5$ (sec). Therefore, for each binary executable, at least three contributors have to get involved in uploading the complete sample. Each user can make at most three attempts to contribute to the same malware program.

We design three scenarios to evaluate the performance of the blockchain-based binary executable sharing scheme:

- **Case 1**: all six clients are cooperative by following the protocol in Algorithm 1 strictly;
- **Case 2**: three clients are fully cooperative but the other three are only partially cooperative: they initiate the transactions to upload their chunks so these transactions appear on the blockchain but they do not attempt to transfer their chunks to BigBing successfully;



(1) Case 1 (2) Case 2 (3) Case 3

Fig. 4. The number of blocks per state on the blockchain maintained on the server

- **Case 3**: three clients are fully cooperative but the other three are malicious and collude among themselves: for each binary executable hash they see, they use the same different file in computing the chunks and their hashes.

Figure 4 shows the number of blocks per state on the blockchain maintained by the server in the three cases. In Case 1, there are 560 binary executables that are shared by at least three of the six fully cooperative users, and they are received successfully by the server in 621 seconds (about 0.17 hour). In this case, because all six clients are fully cooperative, it takes only a short period of time to upload all the sharable binary executables. The two curves representing blocks in the `PENDING` and `COMMITTED` state, respectively, overlap with each other because all the cooperative clients upload the chunks that they have promised to send.

In Case 2, there are 76 binary executables shared by the three fully cooperative users, and it takes 10,985 seconds (about 3.1 hours) for the BigBing server to reassemble all these samples (which correspond to the 76 blocks in a `COMPLETED` state on the blockchain). From Figure 4(2) we observe that blocks in a `PENDING` state outnumber those in a `COMMITTED` state, because there are three partially cooperative users who promise to send their chunks but do not upload them later. There are also three notable bumps for the number of blocks in a `PENDING` state on the blockchain. The first one occurs at the time after around 556 seconds, when each client has finished initiating the transactions for uploading its binary executable chunks *without waiting for any timer to expire* (see Algorithm 1). The second bump happens at the time after around 5444 seconds, when clients wait for a period of $(1 + 1/2) \cdot T_{exp}$ (i.e., 5400 seconds) to recheck an unexpired chunk in a `PENDING` state in the blockchain. The last bump occurs after an elapse time of 10,868 seconds when another such waiting period has expired.

In Case 3, there are 76 binary executables which are shared by three fully cooperative users, and all of them have been received within 675 seconds (about 0.19 hour). Similarly to Case 1, the two curves representing blocks in `PENDING` and `COMMITTED`, respectively, overlap with each other. This is because the three malicious clients use a different file to calculate chunk hashes and after these chunks are uploaded to the server successfully, the states of these chunks are changed from `PENDING` to `COMMITTED` using newly created blocks on the blockchain. Only after the server fails to reassemble a binary executable with a matching hash from these chunks can it realize that these clients have uploaded a different file. Comparing Case 3 against Case 2, we notice that the three malicious and colluding clients affect little the time needed for the three cooperative users to upload the 76 sharable binary executables. This is because these cooperative users know that the blocks created by the malicious ones contain wrong chunk hashes and thus ignore them according to the protocol.

(A.2) PSI operation in user data distribution matching. BigBing uses a PSI-based protocol to ensure client's privacy in uploading her list of binary executable hashes and in this set of experiments, we measure the execution overhead of finding the intersection of binary executable MD5 hashes between the client and the server using two different PSI algorithms, one based on the Diffie-Hellman algorithm [35] and the other oblivious transfer (OT) [38]. In our experiment, the server has 15,983 samples and the client has a local list of 10,000 executable programs with the number of intersecting samples with the server ranging between 2,000 and 10,000. We adapt the implementation code of both algorithms from [18] to run in a distributed computing environment where the client and the server are located on different physical machines. The execution performances using two desktops in our laboratory are depicted in Figure 5, from which it is observed that the OT-based algorithm leads to negligible execution overhead, which is around half a second in all five scenarios.

(A.3) Responsiveness of malware classification modeling as a service. Based on its available computational resources, BigBing uses parameter m to control the size of the training dataset T. We show how the execution times at different stages of BigBing vary with parameter m. In the experiments, we use all different types of Windows PE malware features when training a classifier, and the size of the user-provided sample hash list is 1,000. For each setting of parameter m, we run the experiments for 10 times, and calculate the average execution time of each of the following stages:

- *Transmission*: Forward the user's request to the BigBing backend and return the model trained to the user.

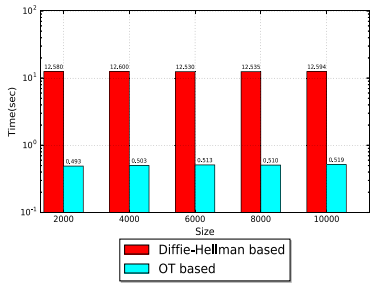


Fig. 5. Execution time of PSI operation in user data distribution matching

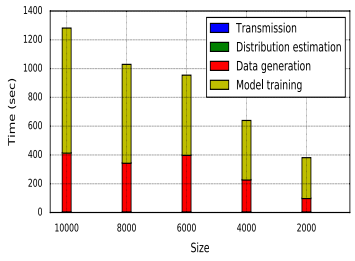


Fig. 6. The execution time of each stage for malware classification as a service

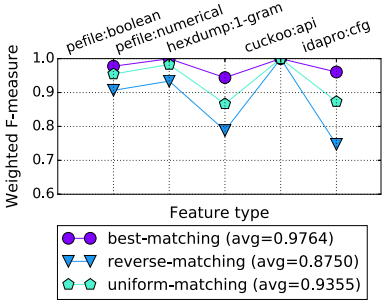


Fig. 7. Performances under different distribution matching schemes

- *Distribution estimation*: Given the user-provided sample hash list, estimate distribution $\mathcal{D}_H$.
- *Data generation*: Based on the user's operational constraints, generate a list of feasible samples, sample this list to get a training dataset of size m, fetch the feature data from the HDFS, and generate the training data file.
- *Model training*: Use the grid search method to find the predictive model that optimizes the user's performance objective while satisfying her performance constraints.

Figure 6 shows the average execution time of each stage after the user submits her request. Clearly, the overall response time is dominated by the data generation and model training stages. When decreasing the training dataset size m, the execution time decreases almost linearly. Even with $m = 10,000$, the overall response time is around 22 minutes, suggesting that the users do not have to wait for a long time to get a predictive model trained by BigBing.

B. Classification performance of BigBing

In this part, we evaluate the classification performances of the prediction models trained by BigBing using the metrics defined in the Appendix. When the classification results based on `cuckoo:api` features are presented, the Internet access mode is used by default unless stated otherwise.

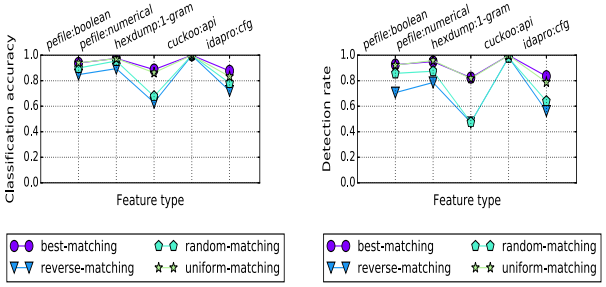
(B.1) Matching users' data distribution. In this set of experiments, we show the importance of matching the malware data distribution in the user's environment when generating the training datasets. We consider the following distribution matching schemes based on $\mathcal{D}_H$, assuming that its malware families are ordered in decreasing order of frequencies:

- **Best-matching:** Based on the user-provided sample hash list H, we generate a training dataset T whose distribution exactly matches $\mathcal{D}_H$ discovered from H. This is the scheme used by BigBing in its operation.
- **Reverse-matching:** We match $\mathcal{D}_H$ except that the order of malware families are reversed. Hence, the family with the least samples in H has the largest number of samples in the training dataset generated.
- **Random-matching:** We match $\mathcal{D}_H$ but randomly change the order of malware families.
- **Uniform-matching:** We generate a training dataset with the same number of samples for each malware family. In this case, the user does not need to provide a representative sample hash list H.

When matching a specific distribution, BigBing may not have sufficient samples for certain underrepresented families. We sample malware features with replacement in distribution matching. Typically, the number of malware families seen in a user's operating environment is much smaller than that available on a cloud-based environment like BigBing. In this set of experiments, we thus assume that the user observes only attacks from a subset of malware families.

We first randomly choose 2,000 unpacked PE samples among all available data, and then select only those from the Bagle, Bifrose, and Hupigon families for testing. On average, about 819 samples are selected for testing. The selected test samples are divided into five folds, and in each experiment, one fold is selected as the user's sample hash list and the remaining ones for testing. The size of the training dataset m is set to be 4,000, and the weighted F-measure is used as the performance objective.

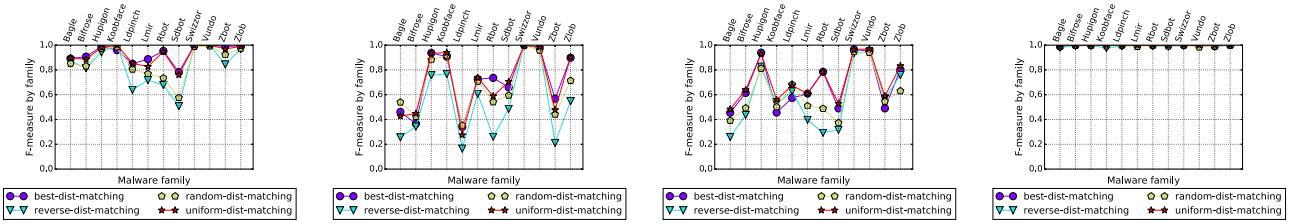
Figure 7 shows the classification performances under different data distribution matching schemes. Since we have only three families in the testing dataset, we ignore the results from random-matching. We observe that the average weighted F-measure across five different feature types using best-matching leads to 11.6% improvement over that under reverse-matching, and 4.4% improvement over that under uniform-matching.



(1) ACC (2) DR under $FPR < 0.05$

Fig. 8. Evaluation results under two different performance objectives, ACC, and DR under $FPR < 0.05$

(B.2) User-specified objective function and constraints. In addition to weighted F-measure, BigBing also supports the



(1) pefile:numerical (2) pefile:boolean (3) hexdump:1-gram (4) cuckoo:api

Fig. 9. The F-measures by family when only packed PE malware samples are considered

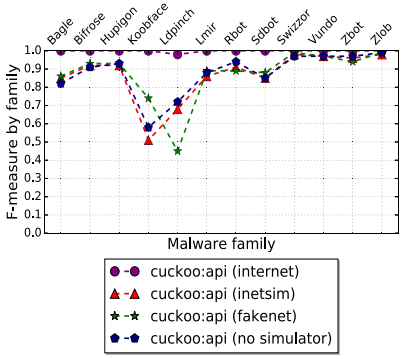


Fig. 10. Classification performances from cuckoo API features under different network configurations

other types of performance objectives and constraints. In a new set of experiments, we use 5,000 malware samples for testing, which are randomly drawn from the unpacked PE malware samples available to BigBing. These test samples are divided to five folds, each of which is used as the user's sample hash list and the remaining for testing. In Figure 8, we show the evaluation results under two different performance objectives, one using classification accuracy (ACC) as the performance objective, and the other detection rate (DR) while under the constraint that the false positive rate should be below 5%. For this set of experiments, we can see that the best-matching scheme performs slightly better than uniform-matching, but does much better than reverse-matching and random-matching. Note that in these experiments, the testing malware samples are drawn from all families, instead of only a subset of them. This leads to smaller improvement of best-matching over uniform-matching, compared with what we see in Figure 7.

(B.3) Packed PE malware samples. BigBing is capable of training predictive models for classifying packed Windows malware. Tools such as PEiD [17] can be used to detect common malware packers. We run a set of experiments with 6,480 packed Windows malware, among which 1,000 are randomly chosen for testing and are thus not seen by BigBing for training the classifiers. The 1,000 test samples are further divided into five folds, each of which is used for user-provided MD5 hash list and the remaining ones to evaluate the classification performances.

Figure 9 shows the average F-measures of each malware family on different types of malware features. From Figure 9, we make the following observations. (1) The API call features extracted from Cuckoo outputs have the best discriminative power. This is expected because features extracted from static malware analysis become less meaningful after malware samples are packed. Among the three types of static malware features, numerical ones extracted from PE headers lead to good classification performances, which is encouraging because the overhead of extracting such features is much lower than setting up a Cuckoo sandbox for dynamic malware analysis. (2) The best-matching scheme leads to the best classification measures compared to the other schemes. This is expected because BigBing trains a classifier that optimizes the classification measure with a training dataset reflecting what is seen in the user's working environment.

(B.4) Effect of network configuration. In another set of experiments, we compare the classification performances based on Cuckoo API features under four different network configurations, including Internet, simulated Internet with InetSim, simulated Internet with FakeNet, and no simulated Internet. During the process of feature extraction, we observe that in some cases Cuckoo does not report any dynamic analysis results. For fair comparison, we consider a set of 15,759 unpacked binary executables with Cuckoo API features extracted successfully. The classification performances under these four different network configurations are shown in Figure 10. We observe that, with Internet access, the classification performance is close to perfection and for the other three network configuration modes, the malware classification performances are comparable. The result shows that further research is needed to improve the realism of Internet simulators used for malware analysis.

VIII. Conclusions

In this work, we have developed a new platform called BigBing which provides a privacy-preserving malware classification service. BigBing uses a blockchain-based method to achieve privacy-preserving sharing of binary executable files. With user-provided operational constraints and performance objective functions, BigBing trains classification models specifically tailored to their operational environments. BigBing relies on big data computing platforms to enhance responsiveness of its malware classification modeling service. Using real-world Windows PE malware samples, we have shown that BigBing offers a promising big data-based platform to fight against the ever-evolving malware threats.

ACKNOWLEDGMENT

We acknowledge NSF Award CNS-1618631 for supporting this work and anonymous reviewers for their constructive comments.

REFERENCES

[1] A simple Python LOC counter. https://github.com/tsaulic/pycount.
[2] About VirusTotal. https://www.virustotal.com/en/about/about/.
[3] Anubis. http://anubis.iseclab.org/.
[4] Apache Cassandra. http://cassandra.apache.org/.
[5] Apache Hadoop. http://hadoop.apache.org/.
[6] Apache Spark. http://spark.apache.org/.
[7] bitcoin. https://bitcoin.org/.
[8] Carbon Black may be leaking terabytes of customer data (UPDATED). http://www.healthcareitnews.com/news/carbon-black-may-be-leaking-terabytes-customer-data-updated.
[9] Cuckoo Sandbox. https://cuckoosandbox.org.
[10] ethereum. https://www.ethereum.org/.
[11] FakeNet. https://practicalmalwareanalysis.com/fakenet/.
[12] Flask. http://flask.pocoo.org/.
[13] hexdump. https://en.wikipedia.org/wiki/Hex_dump.
[14] IDA Pro. https://www.hex-rays.com/.
[15] INetSim: Internet Services Simulation Suite. http://www.inetsim.org/.
[16] pefile. https://github.com/erocarrera/pefile.
[17] PEiD. https://www.aldeid.com/wiki/PEiD.
[18] Private Set Intersection (PSI). https://github.com/encryptogroup/PSI#private-set-intersection-psi.
[19] Scikit-learn: machine learning in Python. http://scikit-learn.org/.
[20] VirusTotal. https://www.virustotal.com/.
[21] Zeus source code 2.0.8.9. https://github.com/Visgean/Zeus.
[22] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *Recent Advances in Intrusion Detection*. Springer, 2007.
[23] R. Benzmller. Malware trends 2017. https://blog.gdatasoftware.com/2017/04/29666-malware-trends-2017.
[24] E. Chung. Antivirus software is 'increasingly useless' and may make your computer less safe. http://www.cbc.ca/news/technology/antivirus-software-1.3668746, 2016.
[25] J. Daemen and V. Rijmen. The design of Rijndael:AES—the advanced. *Journal of Cryptology*, 4(1):3–72, 1991.
[26] C. Dong, L. Chen, and Z. Wen. When private set intersection meets big data: an efficient and scalable protocol. In *Proceedings of the 2013 ACM Conference on Computer & Communications Security*, 2013.
[27] M. J. Freedman, K. Nissim, and B. Pinkas. Efficient private matching and set intersection. In *International conference on the theory and applications of cryptographic techniques*, pages 1–19. Springer, 2004.
[28] X. Hu, K. G. Shin, S. Bhatkar, and K. Griffin. Mutantx-s: Scalable malware clustering based on static features. In *USENIX Annual Technical Conference*, pages 187–198, 2013.
[29] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of ACM conference on Computer and communications security*, 2011.
[30] W. Jansen and T. Grance. Sp 800-144. guidelines on security and privacy in public cloud computing. 2011.
[31] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4), 2011.
[32] P. Li, L. Liu, D. Gao, and M. K. Reiter. On challenges in evaluating malware clustering. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2010.
[33] S.-T. Liu, H.-c. Huang, and Y.-M. Chen. A system call analysis method with MapReduce for malware detection. In *International Conference on Parallel and Distributed Systems*. IEEE, 2011.
[34] M. M. Masud, T. M. Al-Khateeb, K. W. Hamlen, J. Gao, L. Khan, J. Han, and B. Thuraisingham. Cloud-based malware detection for evolving data streams. *ACM Transactions on Management Information Systems*, 2011.
[35] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *IEEE Symposium on Security and Privacy*, pages 134–134. IEEE, 1986.
[36] A. Mohaisen, O. Alrawi, and M. Mohaisen. Amal: High-fidelity, behavior-based automated malware analysis and classification. *Computers & Security*, 52:251–266, 2015.
[37] R. O'Callahan. Disable Your Antivirus Software (Except Microsoft's). https://blog.gdatasoftware.com/2017/04/29666-malware-trends-2017.
[38] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on ot extension. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):7, 2018.
[39] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.
[40] P. Rindal and M. Rosulek. Malicious-secure private set intersection via dual execution. *Proceedings of ACM Conference on Computer and Communications Security (CCS'17)*, 2017.
[41] R. L. Rivest. All-or-nothing encryption and the package transform. In *International Workshop on Fast Software Encryption*. Springer, 1997.
[42] M. Sikorski and A. Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. No Starch Press, 2012.
[43] H. Sun, J. Su, X. Wang, R. Chen, Y. Liu, and Q. Hu. Primal: Cloud-based privacy-preserving malware detection. In *Australasian Conference on Information Security and Privacy*, pages 153–172. Springer, 2017.
[44] G. Yan. Finding common ground among experts' opinions on data clustering: With applications in malware analysis. In *Proceedings of the 30th International Conference on Data Engineering*. IEEE, 2014.
[45] G. Yan. Be sensitive to your errors: Chaining neyman-pearson criteria for automated malware classification. In *ACM Symposium on Information, Computer and Communications Security*, 2015.
[46] G. Yan, N. Brown, and D. Kong. Exploring discriminatory features for automated malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 41–61. Springer, 2013.

APPENDIX A: MULTI-CLASS CLASSIFICATION PERFORMANCE METRICS

Let $X = \{\mathbf{x_0}, \mathbf{x_1}, ..., \mathbf{x_{N-1}}\}$ denote the feature data extracted from N testing malware samples, and their true labels are given by $Y = \{y_0, y_1, ..., y_{N-1}\}$, respectively. The prediction results by a multi-class classifier for the n malware samples are given by $\hat{Y} = \{\hat{y}_0, \hat{y}_1, ..., \hat{y}_{N-1}\}$. The set of malware family labels is denoted by L. We also have the delta function $\delta(p)$ defined, which returns 1 if predicate p is true or 0 otherwise. The following multi-class classification performance metrics can be defined:

- *Classification accuracy* (ACC):

$$ACC = \frac{1}{N}\sum_{i=0}^{N-1}\delta(y_i = \hat{y}_i) \tag{2}$$

- *Weighted F-measure* (F_w):

$$F_w = \frac{1}{N}\sum_{l\in L}F(l)\sum_{i=0}^{N-1}\delta(y_i = l), \tag{3}$$

where $F(l)$ is the F-measure by label defined by

$$F(l) = \frac{2\cdot precision(l)\cdot recall(l)}{precision(l) + recall(l)},$$

with

$$precision(l) = \frac{\sum_{i=0}^{N-1}\delta(\hat{y}_i = l)\delta(y_i = l)}{\sum_{i=0}^{N-1}\delta(\hat{y}_i = l)} \tag{4}$$

$$recall(l) = \frac{\sum_{i=0}^{N-1}\delta(\hat{y}_i = l)\delta(y_i = l)}{\sum_{i=0}^{N-1}\delta(y_i = l)} \tag{5}$$

- *(Weighted) detection rate (DR):*

$$DR = \sum_{l\in L}\frac{\sum_{i=0}^{N-1}\delta(y_i = l)}{N}\cdot recall(l) \tag{6}$$

- *(Weighted) false positive rate (FPR):*

$$FPR = \sum_{l\in L}\frac{\sum_{i=0}^{N-1}\delta(y_i = l)}{N}\cdot\frac{\sum_{i=0}^{N-1}\delta(y_i \neq l)\cdot\delta(\hat{y}_i = l)}{\sum_{i=0}^{N-1}\delta(y_i \neq l)}$$