# HashMTI: Scalable Mutation-based Taint Inference with Hash Records

Xiangdong Kong*, Yong Tang*‡, Pengfei Wang*, Shuning Wei†‡, Tai Yue*
*College of Computer, National University of Defense Technology, Changsha, China
Email: {kongxiangdong, ytang, pfwang, yuetai17}@nudt.edu.cn
†College of Information Science and Engineering, Hunan Normal University, Changsha, China
Email: weishuning@hunnu.edu.cn

***Abstract*—Mutation-based taint inference (MTI) is a novel technique for taint analysis. Compared with traditional techniques that track propagations of taint tags, MTI infers a variable is tainted if its values change due to input mutations, which is lightweight and conceptually sound. However, there are 3 challenges to its efficiency and scalability: (1) it cannot efficiently record variable values to monitor their changes; (2) it consumes a large amount of memory monitoring variable values, especially on complex programs; and (3) its excessive memory overhead leads to a low hit ratio of CPU cache, which slows down the speed of taint inference. This paper presents an efficient and scalable solution named HashMTI. We first explain the above challenges based on 4 observations. Motivated by these challenges, we propose a hash record scheme to efficiently monitor changes in variable values and significantly reduce the memory overhead. The scheme is based on our specially selected and optimized hash functions that possess 3 crucial properties. Moreover, we propose the *DoubleMutation* strategy, which applies additional mutations to mitigate the limitation of the hash record and detect more taint information. We implemented a prototype of HashMTI and evaluated it on 18 real-world programs and 4 LAVA-M programs. Compared with the baseline OrigMTI, HashMTI significantly reduces the overhead while having similar accuracy. It achieves a speedup of 2.5X to 23.5X and consumes little memory which is on average 70.4 times less than that of OrigMTI.**

***Index Terms*—program analysis, taint analysis, software testing**

## I. Introduction

Taint analysis is a well-known technique in software security for tracking information flows. It has many uses such as malware analysis [1] [2] [3] and vulnerability discovery [4] [5]. Especially, dynamic taint analysis (DTA) allows researchers to reason about actual executions, and thus perform precise analysis [6]. During program execution, DTA marks a variable as tainted if the variable depends on data derived from predefined taint sources, such as external inputs. The byte-level DTA, which reports dependencies between input bytes and "interesting" program variables (e.g., unexplored branch conditions), is also used to optimize software testing. For instance, it answers the classical question "where to mutate" [7] [8] of fuzzing and guides concolic execution to selectively symbolize input data [9] [10]. However, DTA suffers from over-taint and under-taint issues since it tracks possible information flows according to predefined propagation rules. It also has considerable overhead for interpreting instructions and storing taint labels.

Compared with traditional taint analysis, another technique works by detecting causal relationships between variable value changes and input mutations [11], which we call "Mutation-based Taint Inference" (MTI). MTI infers a variable is tainted if the variable's values change due to input mutations. This is conceptually sound since a counterfactual causality [12] proves the existence of information flow from the input to the variable. It is also immune to the issues of DTA caused by implicit flows [13] and external calls since it does not track specific paths of information flows. In addition, MTI requires neither predefined propagation rules nor instruction interpretations and thus is lightweight and easy to customize. It is also natively suitable for optimizing software testing since its mutation stages can be integrated into the test case generation process of software testing [14] [15].

However, MTI needs to record the runtime variable values to monitor their changes, which leads to 3 challenges to its efficiency and scalability. First, MTI cannot efficiently record those variable values. According to our observations in Section II-B, existing MTI methods either waste much memory or lose records. Second, for complex code (e.g., deep loops), MTI needs to record a large number of variable values and thus consumes a large amount of memory. Finally, the excessive memory overhead of MTI also leads to a low hit ratio of CPU cache and thus slows down the speed of taint inference.

This paper presents an efficient and scalable solution named HashMTI to resolve the above challenges. For each variable, we compute the hash of its runtime values on the fly to monitor their changes. Thus, each variable has only one hash record and the total number of hash records is a constant. Therefore, we can efficiently monitor changes in variable values with $O(1)$ space complexity and make better use of the CPU cache. To compute the hash record, we select two non-cryptographic hash functions that possess 3 crucial properties (i.e., iterative computations, lightweight and collision resistance). We further optimize them for higher speed. Moreover, we propose the *DoubleMutation* strategy for the mutation stage of MTI, which is designed for mitigating the limitation of the hash record and also helps detect more taint information. The strategy is economically applied based on a heuristic indicator.

‡ Corresponding authors

We implemented a prototype of HashMTI and evaluated it with the original MTI tool OrigMTI on 18 real-world programs and the LAVA-M benchmark [16]. According to the results, HashMTI significantly reduces the memory and time overhead of OrigMTI while maintaining a considerable degree of accuracy. Compared with the OrigMTI, HashMTI consumes 2 to 594.7 times less memory and achieves a speedup of 2.5X to 23.5X. When both are compared with the Libdft [17], HashMTI achieves a similar recall ratio as OrigMTI.

In summary, this paper makes the following contributions:

- **Challenges**. We summarize 3 challenges of existing MTI according to our observations on its recording schemes.
- **Hash record scheme**. We propose a hash record scheme to resolve the aforementioned challenges. It is achieved by our specially selected and optimized hash functions that possess 3 crucial properties we summarized.
- ***DoubleMutation* strategy**. We propose the *DoubleMutation* strategy for the mutation stage of MTI. This strategy can mitigate the limitation of the hash record and help detect more taint information (3% to 35% of the total). We also employ a heuristic indicator to be more efficient.
- **Tool**. We implement a prototype of HashMTI and evaluate it on 18 real-world programs and the LAVA-M benchmark. The results showed that HashMTI can significantly improve the efficiency and scalability of the original MTI.

## II. Background

### A. Mutation-based Taint Inference

Mutation-based taint inference (MTI) is a novel technique for taint analysis. To the best of our knowledge, MTI was first presented in MUTAFLOW [11] for detecting information flow in android applications. Inspired by *mutation analysis* [18], MUTAFLOW directly mutates runtime values returned by sensitive sources to assess whether the mutation changes the values passed to sensitive sinks. In effect, it infers the dependencies between inputs and variables based on causal relationships between input mutations and variable changes.

Other works such as SLF [14] and GREYONE [15] employ MTI to optimize fuzzing, where MTI is also named "Fuzzing-driven Taint Inference" (FTI). According to their papers, the pseudocode of MTI is shown in Algorithm 1. At line 1, the instrumented program is executed to record original variable values (i.e., the operand of branch instructions). At lines 2–4, it systematically mutates each input byte (one at a time) and executes the program again to obtain new records. At lines 5–13, it compares the records of the two executions. If a variable's value changes while an input byte is mutated, it reports that the former is tainted and depends on the latter.

Below, we further discuss the core components of MTI: the mutation strategies and the recording schemes.

**Mutation.** Existing works adopt various mutation strategies. MUTAFLOW directly mutates variable values; however, it may violate input-validation conditions. By contrast, SLF and GREYONE mutate the input data before it is loaded by the program. Specifically, GREYONE mutates each byte several

**Algorithm 1** Mutation-based taint inference

```
Input: $input$, program $\mathbb{P}$ and instrumented program $\mathbb{P}'$
Output: $br.taint[input] \mid br \in branches(\mathbb{P})$
 1: $Record$ = Execute($\mathbb{P}'$, input)
 2: for each position $pos$ in the input do
 3:     $input'$ = Mutate($input, pos$)
 4:     $Record'$ = Execute($\mathbb{P}', input'$)
 5:     for $br \in branches(\mathbb{P})$ do
 6:         $N$ = MIN($len(Record[br]),\ len(Record'[br])$)
 7:         for $i$ from 0 to $N-1$ do
 8:             if $Record[br][i].opnds \neq Record'[br][i].opnds$ then
 9:                 $br.taint[input] \cup= \{pos\}$
10:                 break
11:             end if
12:         end for
13:     end for
14: end for
```

```c
while(1) {
    char c = INPUT_BYTE(fd);
    if (c == 0xD8)
        get_soi(fd); // SOI: start of file
    else if (c == 0xC0 || c == 0xC1 || ...)
        get_sof(fd); // SOF: start of frame
    else if (c == 0xC4)
        get_dht(fd); // DHT: huffman table
    ...
    else if (c == 0xD9)
        return JPEG_REACHED_EOI; // End of Image
    else exit(); // Invalid marker
}
```

Listing 1: Simplified read_marker() of jdmarker.c

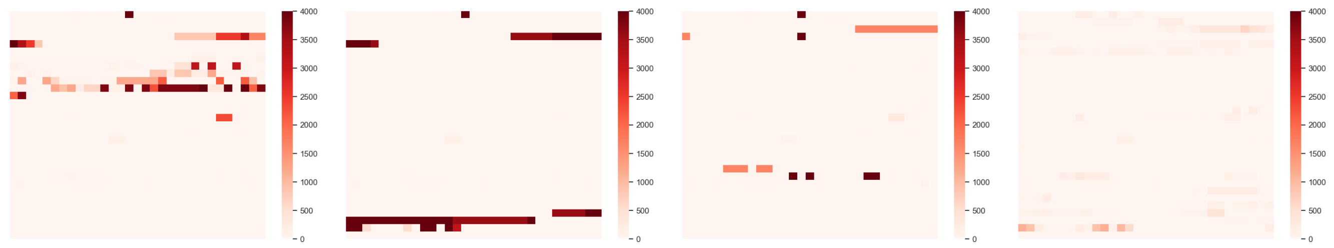
times with a set of strategies, while SLF mutates each byte only once with the operation of "*ByteFlip*". Thus, the mutation stage of MTI can be integrated into the test case generation process. In our opinion, the former can mitigate the influence of non-injective functions (e.g., shift operation) which could conceal the effect of input mutations. However, according to a general consensus [19] [20], the latter is more efficient since it spends less time on systematically mutating input data.

**Recording.** Existing MTI methods record each variable's runtime values to monitor their changes. SLF employs a linear array for recording. It uses program counters to identify each variable, whereby it can align two executions' records to detect the changes in each variable's values. GREYONE assigns a unique ID for each variable and stores its values in a bitmap (with the ID as key). However, both the schemes have their limitations. We discussed these limitations in Section II-B.

### B. Motivation

As shown in Listing 1, we use the code simplified from libjpeg-turbo [21], a known JPEG image codec, to explain the challenges of MTI. We first state 4 key observations on the records (i.e., the variables' runtime values) of MTI as follows.

**A variable has multiple records.** Programs usually use loops to parse the same type of AST nodes. For example,



(a) test case id: 000417 (b) test case id: 000536 (c) test case id: 000642 (d) test case id: 001046

Fig. 1: The heat maps depict the branch execution count of libjpeg-turbo on 4 test cases. Each cell represents a branch and its color indicates the branch's execution count. For simplicity, we omit branches that are not executed in all the 4 test cases.

TABLE I: Memory consumption of MTI when working on timeout test cases of fuzzing ($0.2\ s < exec\_time < 3\ s$).

| Programs | #Testcases | Average(GB) | Maximum(GB) |
|---|---|---|---|
| avconv | 7 | 9.9 | 28.3 |
| ffmpeg | 57 | 7.6 | 20.7 |
| pdftops | 69 | 6.7 | 22.5 |
| djpeg | 9 | 2.9 | 10.0 |
| magick | 63 | 0.2 | 3.3 |
| objdump | 72 | 0.1 | 3.4 |

the code in Listing 1 repeatedly executes the loop to parse the marker of each segment in the JPEG header. To detect information flows from related input bytes to the operand `c`, we should record the operand's values each time it is used for comparison. Thus, the operand of each branch instruction has multiple records whose number is equal to the branch execution count. Considering this phenomenon, in Algorithm 1, we check all records of each variable (at lines 6–12). However, neither SLF nor GREYONE mentioned this phenomenon.

**The number of records is non-uniformly distributed among variables.** As shown in Fig. 1, we used $30 \times 30$ heat maps to show the distribution of branch execution count in the application `libjpeg-turbo`. The distribution of the number of records among variables is also indicated since the number of records is equal to the branch execution count. We can see most branches are rarely executed, while several branches are frequently executed, which reveals a non-uniform distribution. Moreover, the distribution changes a lot for different inputs.

**The total number of records is inconstant.** We also observed that the total number of records is different for different inputs or programs. Moreover, according to our evaluations of the original MTI in Section V-B, those numbers have large variances. The reason is that the total number of records is equal to the total branch execution count, which changes considerably for different execution paths.

**Excessive memory is consumed.** We further observed that MTI consumes a large amount of memory. According to our experiments in Section V-B, the memory consumption is far more than the size of 64 KB suggested by AFL (a state-of-the-art fuzzer that also employs shared memory) [22]. We also collected a set of timeout test cases from our previous fuzzing. Their execution time is 0.2 to 3 seconds, which means that they are executing complex code but not endless loops. As shown in Table I, the consumption of MTI becomes more excessive when working on these test cases.

Based on these observations, we discussed the limitations of existing MTI methods. First, neither SLF nor GREYONE can efficiently record runtime values of variables. The bitmap used by GREYONE is especially unsuitable since we do not know how much space should be reserved for each variable. For example, AFLplusplus [23] uses a $65536 \times 256$ bitmap for similar purposes as MTI, which reserves 256 items for multiple records of each variable. However, the size of 256 will both waste much memory and lose many records according to the aforementioned observations. It is also hard to determine how much memory should be allocated for the linear array of SLF. Besides, the scheme of SLF consumes much time aligning two executions' records. Second, the excessive memory consumption is inevitable since MTI attempts to record concrete variable values. Finally, although memory overhead is no longer a serious problem, it leads to a low hit ratio of CPU cache and thus slows down the taint inference [24]. We summarized these limitations into 3 challenges that seriously limit the efficiency and scalability of MTI, as follows.

**Challenge 1.** *MTI cannot efficiently record runtime values of the variables to monitor their changes.*

**Challenge 2.** *MTI consumes large amounts of memory recording variables' runtime values, especially on complex code.*

**Challenge 3.** *The memory overhead of MTI leads to a low hit ratio of CPU cache and thus slows down the taint inference.*

Therefore, we need a solution that can improve the efficiency and scalability of MTI while maintaining its accuracy.

## III. Methodology

To address the aforementioned challenges, we proposed an efficient and scalable MTI method named HashMTI. Below, we elaborate on the details of this method.

### A. Overview

Fig. 2 illustrates the workflow of HashMTI, which contains two main components: *the compile phase for instrumentation and the main module for taint inference*. Static analysis on LLVM IR is initially performed to obtain information of the target program. Based on this information, a unique position in the bitmap is assigned for each variable (i.e., operands

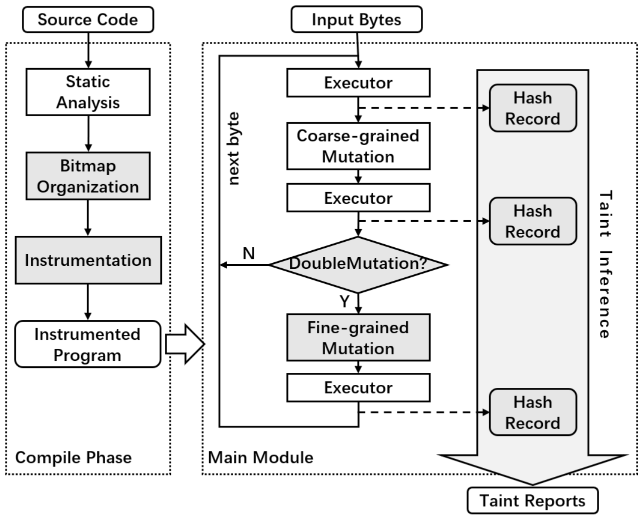


Fig. 2: Overview of HashMTI. The grey shapes denote key processes of our approach.

of branch instructions) to store their records. Based on the results of the static analysis and bitmap organization, we insert instrumentations for our key approaches: the hash record scheme (see Section III-B) and the *DoubleMutation* strategy (see Section III-C).

After the program is instrumented, HashMTI enters the main module for taint inference. First, similar to Algorithm 1, it runs the program with a given input to obtain the original record $Record$. Then, it mutates each input byte with the coarse-grained mutator (one at a time). Once a byte is mutated, it runs the program again to obtain the new record $Record^{'}$. Next, it uses a heuristic indicator to decide whether to mutate the byte again with the fine-grained mutator. The execution after the fine-grained mutation also has a new record $Record^{''}$. Based on these records, HashMTI infers which variable is tainted and depends on the current byte. After the processing on the current byte, HashMTI goes through the remaining input bytes and repeats the above processes.

Below, we detail our hash record scheme and *DoubleMutation* strategy, which are the key approaches of HashMTI.

## *B. Hash Record Scheme*

Considering the challenge 1, the linked list structure can be used to allocate appropriate memory for each variable to store its multiple records. However, the appending operation is complex, and the runtime memory allocation has extra overhead. Motivated by the linked list, we further found that there exist such hash functions that can compute the hash of a sequence iteratively. Later items of the sequence are combined with the hash of previous items, which is similar to the appending operation of a linked list.

Based on the above facts, we propose the hash record scheme. For each variable to monitor, we regard its runtime values as an ordered sequence whose items successively arrive during the program execution. We iteratively compute the sequence's hash as the hash record of each variable, whereby

```
1  while(1) {
2    char c = INPUT_BYTE(fd);
3    // id: assigned position of variable c
4    if (hash_record[id].cnt < hash_record[id].UB) {
5      ++hash_record[id].cnt;
6      old_hash = hash_record[id].hash;
7      updated_hash = HashFunc(c, old_hash);
8      hash_record[id].hash = updated_hash;
9    }
10   if (c == 0xD8)
11     get_soi(fd); // SOI: start of file
12   ...
13 }
```

Listing 2: Example of HashMTI's instrumentations. The code is from Listing 1, and the instrumentations are highlighted.

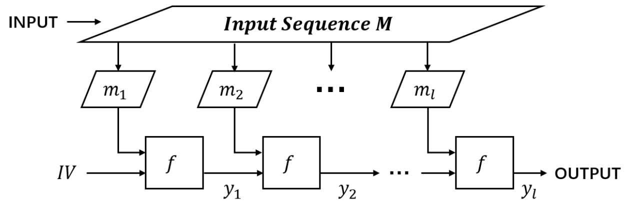


Fig. 3: Merkle–Damgard construction.

we can still detect the changes in variable values (according to the hash checksum) while reducing the space complexity of MTI to $O(1)$. Therefore, we can efficiently store the hash records in a bitmap with the space complexity of $O(1)$, thereby resolving the challenges described in Section II-B.

We further illustrate our scheme with Listing 2. The code is from Listing 1 and instrumented by HashMTI to compute the hash record of variable c at line 10. Each time the variable is used for comparison, its runtime value is used to update the hash record computed in previous iterations (at lines 6–8 in Listing 1). When the program exits, the up-to-data hash is the final hash record of the variable.

Note that not all hash algorithms are suitable for computing hash records. Most well-known hash functions such as MD5 and SHA1 are not suitable since their computations are not iterative. The efficiency and collision resistance are also important for MTI. We summarized 3 crucial properties that a suitable hash function should possess as follows.

**Property 1.** *The function computes the hash of an ordered sequence iteratively with $O(1)$ space complexity. It combines each item of the sequence with the final result one by one without any information of later items.*

**Property 2.** *The function is sufficiently lightweight and easy to implement with the instrumentation.*

**Property 3.** *The function is collision-resistant to sensitively detect the changes in variable values.*

According to these properties, we surveyed existing hash functions and found that *non-cryptographic hash functions* (NCHFs) are especially suitable. The NCHFs are a family of hash functions that are not considered safe but have

**Algorithm 2** DJBX33A hash function

```
Input: sequence /* runtime values of each variable */
Output: hash
1: hash = 5381 /* 8-byte variable */
2: for each item in sequence do
3:     hash = item + hash * 33
4: end for
```

**Algorithm 3** Splitmix64 hash function

```
Input: sequence /* runtime values of each variable */
Output: hash
1: hash = 0 /* 8-byte variable */
2: for each item in sequence do
3:     hash += item
4:     hash = hash xor (hash >> 30)
5:     hash = hash * 0xbf58476d1ce4e5b9
6:     hash = hash xor (hash >> 27)
7:     hash = hash * 0x94d049bb133111eb
8:     hash = hash xor (hash >> 31)
9: end for
```

high performance. They are usually constructed following an iterative scheme, known as *Merkle-Damgard construction*. As shown in Fig. 3, it divides the input sequence $M$ into fixed-size items $(m_1, m_2, ..., m_l)$ and combines them with the internal state $(IV, y_1, ..., y_l)$ one by one using a mixing function $f$ [25]. The space complexity of this process is $O(1)$.

In Table II, we checked 8 common NCHFs according to previous evaluations [25] [26] [27] [28]. Considering property 1, MurmurHash3 is not suitable since it requires the sequence length for computations. Considering property 2, Lookup3 is too complex. In addition, all those functions divide the input sequence into one-byte items. Thus, when used in HashMTI, they have to go through each byte of the variable value whose width is at most 8 bytes on 64-bit machines. For efficiency, we optimized them by extending their item size to 8 bytes; thus, they can combine the variable value with their internal states once and for all. Considering property 3, we accepted most functions though they have collisions in previous evaluations. This is because their collisions are negligible for HashMTI according to the experiments in Section V-E.

We finally selected the optimized DJBX33A and Splitmix64 as the optional hash functions of our scheme (see Algorithm 2 and 3). They are representative and have their own advantages: the former is one of the fastest functions, while the latter has better collision resistance.

## *C. DoubleMutation Strategy*

**The limitation of the hash record scheme.** As mentioned previously, our hash record scheme can resolve the challenges described in Section II-B. However, it has limitations along with its attractive features. For runtime values of each variable, we can determine their changes only if their hash changes while the number of them remains the same. Unfortunately, the number of variable values often changes due to input mutations. As illustrated in Fig. 4a and 4b, the fourth marker in the JPEG header is mutated from `0xffc0` to an undefined value `0xff3f`, which causes the code to exit early. Thus, the number of operand $c$'s runtime values decreases from 5 to 4. In this case, although the hash record changes, we cannot determine whether the runtime values of the variable change.

TABLE II: Properties of non-cryptographic hash functions

| Hash Functions | Property 1 | Property 2 | Property 3 |
|---|---|---|---|
| Apartow [29] | ✓ | ✓ | ✓ |
| MurmurHash3 [30] | × | ✓ | ✓ |
| SuperFastHash [31] | ✓ | ✓ | × |
| DJBX33A | ✓ | ✓ | ✓ |
| One At a Time [32] | ✓ | ✓ | ✓ |
| Splitmix64 [33] | ✓ | ✓ | ✓ |
| Lookup3 [34] | – | × | – |
| FNV [35] | ✓ | ✓ | ✓ |

✓: possess ×: not possess –: unknow

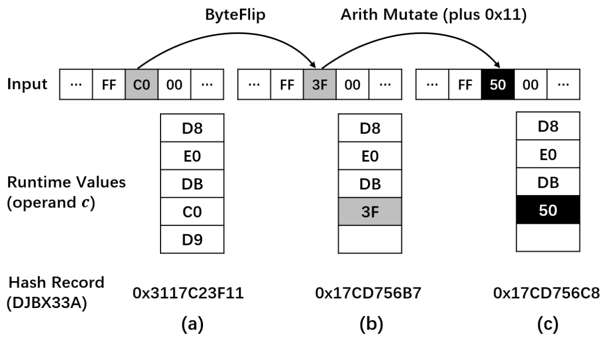


Fig. 4: Example showing that the number of variable values decreases due to mutations (based on the code in Listing 1).

The situation where the number of variable values increases after mutations is easy to handle. For each variable, we use the number of its runtime values before mutations as its upper bound $\lambda$. For the mutated input, we hash only the first $\lambda$ runtime values of the variable as its hash record (see line 4 in Listing 2). Thus, we can omit its extra runtime values. However, the other situation where the number of values decreases is more common since mutations often break the file structure and cause the program to exit early [36]. To handle this situation, we proposed a novel strategy: *DoubleMutation*.

**What is *DoubleMutation*.** As the name implies, the strategy applies another mutation on the mutated input byte. We observed that there are far more invalid values than the valid, and those invalid values are usually handled by a single program state. Thus, mutating the broken byte again usually causes no further decrease in the number of variable values. As illustrated in Fig. 4c, we applied an arithmetical mutation on the already mutated byte `0x3f`, which causes no decrease in the number of variable values. By comparing the hash records in Fig. 4b and Fig. 4c, we can determine their changes because they have the same number of values but different hashes.

The pseudocode of the *DoubleMutation* is shown in Algorithm 4. At line 10, we perform the additional arithmetical mutation. At line 11, we set the aforementioned upper bounds for each variable as a complement to *DoubleMutation*. At lines

**Algorithm 4** HashMTI with *DoubleMutation*

```
Input: $input$, program $\mathbb{P}$ and instrumented program $\mathbb{P}'$
Output: $br.taint[input] \mid br \in branches(\mathbb{P})$
1: SetUpperBound($\infty$)
2: $Record$, $exec\_cksum$ = Execute($\mathbb{P}'$,input)
3: for each position $pos$ in the input do
4:     $input'$ = ByteFlip($input, pos$)
5:     SetUpperBound($Record$)
6:     $Record'$, $exec\_cksum'$ = Execute($\mathbb{P}', input'$)
7:     if $exec\_cksum == exec\_cksum'$ then
8:         ... /* Applying taint inference like original MTI */
9:     else /* DoubleMutation */
10:        $input''$ = ArithMutate($input', pos, 0x11$)
11:        SetUpperBound($Record'$)
12:        $Record''$, $exec\_cksum''$ = Execute($\mathbb{P}', input''$)
13:        for $br \in branches(\mathbb{P})$ do
14:            if $Record[br].cnt == Record'[br].cnt$ && $Record[br].hash$ != $Record'[br].hash$ then
15:                $br.taint[input] \cup = \{pos\}$
16:            else if $Record[br].cnt == Record''[br].cnt$ && $Record[br].hash$ != $Record''[br].hash$ then
17:                $br.taint[input] \cup = \{pos\}$
18:            else if $Record'[br].cnt == Record''[br].cnt$ && $Record'[br].hash$ != $Record''[br].hash$ then
19:                $br.taint[input] \cup = \{pos\}$
20:            end if
21:        end for
22:    end if
23: end for
```

13–21, we check the differences between hash records of the three executions to detect which variable is tainted.

**Mutation operand.** To maintain the same number of variable values, the additional mutation with a small operand is fine-grained. However, the operand should not be too small (e.g., 1), or the effects of input mutations could be concealed by non-injective functions (e.g., shift operation). Intuitively, we select the value `0x11`, which is not affected by common shift operations on one byte (e.g., >>4).

**When to apply *DoubleMutation*.** We do not apply *DoubleMutation* on all input bytes since its additional executions slow down the taint inference. It is also not worth applying *DoubleMutation* if the limitation of the hash record affects only a few variables. To decide when to apply *DoubleMutation*, we used the `exec_cksum` of AFL as a heuristic indicator (see line 7 in Algorithm 4). It measures the influence of input mutations on the branch execution count which is equal to the number of each variable's runtime values. Thus, we can effortlessly determine whether a considerable number of variables are affected, and decide whether to apply the additional mutations.

## IV. Implementation

We implemented a prototype of HashMTI with 2.2k lines of C/C++ and Python code based on the AFL and the LLVM. Here, we present some of its implementation details.

TABLE III: The configuration of target programs to test.

| Subject | Version | Format | #Branches | #Testcases |
|---|---|---|---|---|
| avconv | libav-12.3 | MP4 | 129K | 557 |
| ffmpeg | ffmpeg-4.1.3 | MP4 | 235K | 1824 |
| bsdtar | libarchive-3.4.0 | TAR | 15K | 1,701 |
| cxxfilt | binutils-2.34 | ELF | 26K | 1,297 |
| objdump | binutils-2.34 | ELF | 34K | 2,742 |
| readelf | binutils-2.34 | ELF | 11K | 6,155 |
| readpng | libpng-1.6.37 | PNG | 4,791 | 6,284 |
| djpeg | libjpeg-turbo-2.0.2 | JPEG | 2,945 | 1,060 |
| jhead | jhead-3.04 | JPEG | 745 | 3,316 |
| infotocap | ncurses-6.2 | TEXT | 6,560 | 1,453 |
| tcpdump | tcpdump-4.9.3 | PCAP | 14K | 10,255 |
| xmllint | libxml2-2.9.10 | XML | 36K | 3,498 |
| gif2png | gif2png-2.5.13 | GIF | 386 | 3,669 |
| pdftops | xpdf-4.02 | PDF | 24K | 295 |
| pdfinfo | xpdf-4.02 | PDF | 21K | 834 |
| tiff2pdf | tiff-4.0.10 | TIFF | 6,578 | 993 |
| tiff2bw | tiff-4.0.10 | TIFF | 5,092 | 1,362 |
| tiffinfo | tiff-4.0.10 | TIFF | 5,212 | 1,367 |
| base64 | coreutils-8.24-lava | DATA | 527 | 1,958 |
| who | coreutils-8.24-lava | DATA | 5,705 | 3,962 |
| md5sum | coreutils-8.24-lava | DATA | 583 | 2,542 |
| uniq | coreutils-8.24-lava | DATA | 491 | 412 |

Based on the LLVM IR, HashMTI analyses each branch instruction and assigns each of them a unique position in the bitmap. The bitmap is in the shared memory and its size is the total number of branches multiplied by the size of the record. Note that, for efficiency, we compute the hash of both the two operands' values as the hash record of each branch instruction. Thus, we allocate only $3 \times 8$ bytes for the record of each branch instruction, containing the hash value, the number of operand values (i.e., the branch execution count) and the upper bound.

Moreover, in this paper, we concentrate on the technique of MTI itself. Since the prototype of HashMTI is implemented on top of AFL, we cancel the test case generation process of the fuzzer and leave the taint-guided fuzzing for future works.

## V. Evaluation

In this section, we conduct thorough experiments to evaluate our work HashMTI. With these experiments, we aim to answer the following research questions:

- **RQ1.** Compared with the original MTI, how much memory overhead does HashMTI reduce?
- **RQ2.** Compared with the original MTI, does HashMTI speed up the taint inference?
- **RQ3.** How efficient is the *DoubleMutation* strategy?
- **RQ4.** How much does the limitation of the hash record scheme affect the accuracy of HashMTI?

### A. Experiment Setup

**Target Programs.** We evaluated HashMTI on 18 real-world programs, including tools and libraries used for processing various file formats. We also evaluated HashMTI on the LAVA-M benchmark [16], which has many inserted hardcode comparisons. The configuration of all subjects is listed in Table III. Intuitively, the program with more branches is more complex. Note that the target programs we used are not exactly the same as in the previous work. More large applications were tested to evaluate the efficiency and scalability of HashMTI.

**Testcases.** We ran HashMTI on a set of test cases (i.e., inputs of programs) which are collected during previous fuzzing with state-of-the-art fuzzers and minimized by *afl-cmin*. Table III lists the number of these test cases for each subject.

**Baseline.** Since neither SLF nor GREYONE is open source, we implemented an original MTI tool as the baseline according to Algorithm 1, called OrigMTI. We used a 1 GB linear array for variable value monitoring, which is similar to SLF and sufficient in our experiments. We did not adopt the scheme of GREYONE since it loses records and its performance relies on its closed-source bitmap configurations. In addition, we used an out-of-the-box DTA tool, the Libdft [17], to further evaluate our work. We set its maximum execution time is to 2 minutes and its maximum memory usage to 4 GB.

**Performance Metrics.** To compare the performance of HashMTI and OrigMTI, we consider both their memory and time overhead. We count only the actual usage of OrigMTI's linear array since its size of 1 GB is sufficient but not necessary. The time overhead is evaluated from 3 aspects: (1) the time of executing the instrumented program once to evaluate the overhead of monitoring runtime variable values; (2) the total execution time on each test case to evaluate the extra time consumption of HashMTI introduced by its additional executions; and (3) the total taint inference time of the two tools, including both the overhead of executions and record analysis. In addition, we evaluate the accuracy of HashMTI. Since its accuracy could be affected by the hash collisions and the limitation of the hash record, we use the collision rate, false negative and recall ratio of taint reports to measure the losses in its accuracy. Considering the baseline used in this paper is implemented by ourselves, we also perform similar experiments as those in previous works GREYONE and compare the experimental results.

**Experiment Environment.** The experiments were conducted on a 64-bit machine with 8 cores (2.5 GHz Intel Xeon Platinum 8269CY), 32 GB of RAM, and Ubuntu 18.04 as the server OS. We ran each tool on a single core. When measuring the time consumption, we repeated each experiment 6 times and computed the arithmetic average to reduce the influence of other running processes.

## B. Evaluating the Memory Overhead (RQ1)

For each MTI tool, Table IV reports its memory consumption on each subject. We can see that HashMTI has low memory overhead. On 6 subjects, its consumption is lower than the size of 64 KB suggested by AFL [22]. On average, it consumes only 0.6 MB of memory. Moreover, we use the $Factor$ to indicate how many times the memory consumption of HashMTI is less than that of OrigMTI. As indicated by the factor$_1$, the consumption of HashMTI is 2 to 594.7 times less than the average memory consumption of OrigMTI. As indicated by the factor$_2$, the consumption of HashMTI is at most 101,369 times less than that of OrigMTI. The results show that HashMTI significantly reduces the memory consumption of original MTI on 21 of the 22 subjects. The reason is that our hash record scheme reduces the space complexity of MTI to $O(1)$. On the `tcpdump`, HashMTI consumes more memory because its bitmap reserves storage locations for all branches, though some of them are not executed. Nevertheless, on that subject, it consumes only 0.32 MB of memory, which is far less than the maximum consumption of OrigMTI.

TABLE IV: Memory consumption of HashMTI and OrigMTI.

| Subject | HashMTI* | OrigMTI (Avg) | Factor$_1$ | OrigMTI (Max) | Factor$_2$ |
|---|---|---|---|---|---|
| gif2png | 9.04 KB | 5.25 MB | 594.7 | 894.9 MB | 101369 |
| md5sum | 0.01 MB | 2.45 MB | 245.0 | 121.3 MB | 12130 |
| djpeg | 0.06 MB | 8.14 MB | 135.7 | 92.1 MB | 1535 |
| pdftops | 0.54 MB | 63.23 MB | 117.1 | 637.2 MB | 1180 |
| uniq | 0.01 MB | 1.17 MB | 117.0 | 39.4 MB | 3940 |
| base64 | 0.01 MB | 1.03 MB | 103.0 | 24.8 MB | 2480 |
| infotocap | 0.15 MB | 11.45 MB | 76.3 | 58.6 MB | 391 |
| pdfinfo | 0.48 MB | 27.54 MB | 57.4 | 184.6 MB | 385 |
| tiffinfo | 0.11 MB | 3.13 MB | 28.5 | 34.8 MB | 316 |
| avconv | 2.95 MB | 69.18 MB | 23.5 | 914.7 MB | 310 |
| jhead | 0.01 MB | 0.09 MB | 9.0 | 2.8 MB | 280 |
| tiff2bw | 0.11 MB | 0.81 MB | 7.4 | 73.3 MB | 666 |
| objdump | 0.77 MB | 4.88 MB | 6.3 | 126.1 MB | 164 |
| ffmpeg | 5.37 MB | 32.68 MB | 6.1 | 105.6 MB | 20 |
| tiff2pdf | 0.15 MB | 0.83 MB | 5.5 | 12.3 MB | 82 |
| xmllint | 0.82 MB | 3.04 MB | 3.7 | 78.3 MB | 95 |
| readelf | 0.25 MB | 0.73 MB | 2.9 | 56.9 MB | 228 |
| cxxfilt | 0.59 MB | 1.65 MB | 2.8 | 33.2 MB | 56 |
| who | 0.12 MB | 0.28 MB | 2.3 | 2.5 MB | 21 |
| bsdtar | 0.34 MB | 0.75 MB | 2.2 | 29.1 MB | 86 |
| readpng | 0.10 MB | 0.20 MB | 2.0 | 17.5 MB | 175 |
| tcpdump | 0.32 MB | 0.08 MB | 0.3 | 51.9 MB | 162 |
| **Avg.** | 0.60 MB | 10.84 MB | 70.4 | 163.3 MB | 5731 |

*The memory consumption of HashMTI is a constant.

Based on the results in Table IV, we can positively answer **RQ1** that HashMTI can significantly reduce the memory overhead of OrigMTI. Thus, it scales well to large programs and makes better use of the CPU cache.

## C. Evaluating the Time Overhead (RQ2)

We compare the time overhead of HashMTI and OrigMTI on 9 representative subjects, including both large programs (e.g., `pdftops`, `ffmpeg` and `avconv`) and small programs (e.g., LAVA-M). Following Klees' recommendation [37], we use the Vargha-Delaney statistic ($\hat{A}_{12}$) to determine the probability that our work outperforms OrigMTI [38] [39]. We also use the $Factor$ to measure the performance gain as the mean time of OrigMTI divided by the mean time of HashMTI.

First, Table V reports the execution speed of the two tools. Intuitively, the program instrumented by OrigMTI should be faster since its instrumentations are simpler. However, our experiments showed the opposite results. All the factors are greater than 1.00 and all the $\hat{A}_{12}$ values are 1.00, which indicates the execution speed of HashMTI is always faster. It achieves a speedup of 1.15X to 2.65X, especially on large programs. The reason is that HashMTI consumes less memory and thus can make better use of the CPU cache than OrigMTI. The results are in agreement with those in CollAFL [24].

Second, Table VI reports the comparison of total execution time. We can see the factors are lower than those in Table V, and the $\hat{A}_{12}$ value is smaller than 1.00 on `readelf`. The reason is that the *DoubleMutation* strategy requires additional

TABLE V: Time of executing the instrumented program once.

| **Subject** | **Tools** | **Average Time(ms)** | **Factor** | $\hat{A}_{12}$ |
|---|---|---|---|---|
| avconv | HashMTI | 31.485 | 2.65 | 1.00 |
| | OrigMTI | 83.521 | – | – |
| ffmpeg | HashMTI | 14.411 | 2.62 | 1.00 |
| | OrigMTI | 37.692 | – | – |
| pdftops | HashMTI | 41.350 | 2.44 | 1.00 |
| | OrigMTI | 100.701 | – | – |
| djpeg | HashMTI | 5.561 | 1.98 | 1.00 |
| | OrigMTI | 11.051 | – | – |
| objdump | HashMTI | 9.340 | 1.73 | 1.00 |
| | OrigMTI | 16.204 | – | – |
| readpng | HashMTI | 1.121 | 1.50 | 1.00 |
| | OrigMTI | 1.686 | – | – |
| tcpdump | HashMTI | 1.394 | 1.27 | 1.00 |
| | OrigMTI | 1.783 | – | – |
| readelf | HashMTI | 3.792 | 1.15 | 1.00 |
| | OrigMTI | 4.371 | – | – |
| jhead | HashMTI | 0.498 | 1.15 | 1.00 |
| | OrigMTI | 0.574 | – | – |

TABLE VI: Total program execution time on a test case.

| **Subject** | **Tools** | **Average Time(s)** | **Factor** | $\hat{A}_{12}$ |
|---|---|---|---|---|
| avconv | HashMTI | 74.001 | 2.26 | 1.00 |
| | OrigMTI | 167.305 | – | – |
| ffmpeg | HashMTI | 77.874 | 1.94 | 1.00 |
| | OrigMTI | 151.436 | – | – |
| pdftops | HashMTI | 117.047 | 1.75 | 1.00 |
| | OrigMTI | 205.685 | – | – |
| djpeg | HashMTI | 31.611 | 1.35 | 1.00 |
| | OrigMTI | 42.517 | – | – |
| objdump | HashMTI | 60.469 | 1.05 | 1.00 |
| | OrigMTI | 63.896 | – | – |
| readpng | HashMTI | 5.421 | 1.16 | 1.00 |
| | OrigMTI | 6.317 | – | – |
| tcpdump | HashMTI | 6.605 | 1.06 | 1.00 |
| | OrigMTI | 7.040 | – | – |
| readelf | HashMTI | 16.833 | 1.01 | 0.67 |
| | OrigMTI | 16.972 | – | – |
| jhead | HashMTI | 0.897 | 1.15 | 1.00 |
| | OrigMTI | 1.046 | – | – |

executions on some input bytes and thus introduces extra time overhead. Nevertheless, the factors are still greater than 1.00, and the $\hat{A}_{12}$ values are still considerable, which means that HashMTI is still faster than OrigMTI. On large programs (e.g., `avconv`, `ffmpeg` and `pdftops`), HashMTI can still achieve a speedup of 1.75X to 2.26X. The reason is that the performance improvement of HashMTI due to the better use of CPU cache is significant enough and the *DoubleMutation* strategy is economically applied.

Finally, Table VII reports the comparison of the total taint inference time. The factors considerably increase compared with those of the execution time. Especially, on large programs such as `avconv`, `ffmpeg` and `pdftops`, HashMTI is approximately 8 to 22 times faster than OrigMTI. On average, it achieves a speedup of 9 times. This is because OrigMTI spends much time on loading, aligning and comparing records in the linear array. It checks all runtime values of each variable when analyzing the records (see lines 7–12 in Algorithm 1). By contrast, HashMTI can efficiently leverage a bitmap for storage. Each variable has only one hash record and thus requires only one time comparison (see lines 14–20 in Algorithm 4). In effect, the runtime hash computations of HashMTI reduces the time complexity of the record comparison process to $O(1)$.

TABLE VII: Total taint inference time on a test case.

| **Subject** | **Tools** | **Average Time(s)** | **Factor** | $\hat{A}_{12}$ |
|---|---|---|---|---|
| avconv | HashMTI | 81.389 | 16.7 | 1.00 |
| | OrigMTI | 1364.058 | – | – |
| ffmpeg | HashMTI | 108.879 | 9.05 | 1.00 |
| | OrigMTI | 986.140 | – | – |
| pdftops | HashMTI | 118.067 | 23.5 | 1.00 |
| | OrigMTI | 2782.303 | – | – |
| djpeg | HashMTI | 32.033 | 14.5 | 1.00 |
| | OrigMTI | 462.858 | – | – |
| objdump | HashMTI | 63.019 | 12.9 | 1.00 |
| | OrigMTI | 815.366 | – | – |
| readpng | HashMTI | 5.720 | 4.98 | 1.00 |
| | OrigMTI | 28.491 | – | – |
| tcpdump | HashMTI | 7.547 | 3.18 | 1.00 |
| | OrigMTI | 24.026 | – | – |
| readelf | HashMTI | 17.606 | 2.52 | 1.00 |
| | OrigMTI | 44.356 | – | – |
| jhead | HashMTI | 0.966 | 3.04 | 1.00 |
| | OrigMTI | 2.946 | – | – |

TABLE VIII: The additional execution rates and contribution rates of the *DoubleMutation* strategy.

| **Subject** | **Additional Execution Rate** | | **Contribution Rate** | |
|---|---|---|---|---|
| | Average | Maximum | Average | Maximum |
| avconv | 0.16 | 1.00 | 0.04 | 0.27 |
| ffmpeg | 0.40 | 1.00 | 0.03 | 0.29 |
| pdftops | 0.52 | 0.98 | **0.16** | 0.75 |
| djpeg | 0.40 | 1.00 | **0.35** | 0.90 |
| objdump | 0.53 | 1.00 | 0.11 | 0.72 |
| readpng | 0.41 | 1.00 | **0.29** | 0.67 |
| tcpdump | 0.24 | 0.82 | 0.09 | 0.73 |
| readelf | 0.22 | 0.81 | 0.03 | 0.59 |
| jhead | 0.18 | 0.97 | **0.20** | 0.99 |

Thus, the performance of HashMTI is further improved.

According to the above experiments, HashMTI has significant performance advantages over OrigMTI, especially on large programs (which have a large number of branches).

From the analysis of Table V, VI and VII, we can positively answer **RQ2** that HashMTI can greatly improve the efficiency of MTI and scale well to large programs.

## D. Evaluating the Efficiency of DoubleMutation (RQ3)

As described in Section III-C, the *DoubleMutation* strategy is designed to mitigate the limitation of the hash record scheme. However, it spends extra time on its additional mutations and subsequent executions. We compute its additional execution rate and contribution rate to measure its efficiency. For each test case, the former is the number of additional executions divided by the number of bytes; the latter is the number of taint reports that can be found only by the strategy divided by the number of total reports. Since *DoubleMutation* mutates each byte twice at most, the additional execution rate is no higher than 1.0.

As shown in Table VIII, the average additional execution rates are no higher than 0.52. This means that the strategy introduces only about 0.5 additional executions on average. This is because our heuristic indicator (i.e., the `exec_cksum`) activates additional mutations economically. Moreover, the strategy contributes more than 16% of the taint reports on 4 of

TABLE IX: Hash collision rates of the optimized DJBX33A.

| **Subject** | **Collision Rates of Taint Reports** | **Collision Rates of Branches** | **Collision Rates of Testcases** |
|---|---|---|---|
| who | 0 | 0 | 0 |
| avconv | $1.27 \times 10^{-6}$ | $2.92 \times 10^{-3}$ | $5.02 \times 10^{-2}$ |
| ffmpeg | $8.38 \times 10^{-6}$ | $9.20 \times 10^{-3}$ | $1.09 \times 10^{-1}$ |
| readelf | $1.05 \times 10^{-6}$ | $2.45 \times 10^{-3}$ | $3.50 \times 10^{-3}$ |
| jhead | $2.00 \times 10^{-7}$ | $6.09 \times 10^{-3}$ | $6.00 \times 10^{-4}$ |
| gif2png | $5.45 \times 10^{-6}$ | $6.51 \times 10^{-3}$ | $2.63 \times 10^{-2}$ |
| base64 | 0 | 0 | 0 |
| pdftops | $4.52 \times 10^{-6}$ | $3.02 \times 10^{-2}$ | $8.58 \times 10^{-1}$ |
| tiffinfo | $8.41 \times 10^{-6}$ | $1.54 \times 10^{-2}$ | $4.82 \times 10^{-2}$ |
| tiff2pdf | $3.30 \times 10^{-7}$ | $1.41 \times 10^{-3}$ | $1.00 \times 10^{-3}$ |
| uniq | 0 | 0 | 0 |
| tiff2bw | $2.50 \times 10^{-7}$ | $1.45 \times 10^{-3}$ | $2.20 \times 10^{-3}$ |
| tcpdump | $8.30 \times 10^{-7}$ | $8.77 \times 10^{-4}$ | $1.40 \times 10^{-3}$ |
| djpeg | $3.20 \times 10^{-7}$ | $4.85 \times 10^{-3}$ | $6.90 \times 10^{-3}$ |
| infotocap | $1.99 \times 10^{-6}$ | $1.17 \times 10^{-2}$ | $5.37 \times 10^{-2}$ |
| xmllint | $9.92 \times 10^{-6}$ | $9.63 \times 10^{-3}$ | $2.35 \times 10^{-2}$ |
| pdfinfo | $2.50 \times 10^{-7}$ | $5.77 \times 10^{-3}$ | $2.12 \times 10^{-2}$ |
| objdump | $1.84 \times 10^{-6}$ | $5.69 \times 10^{-3}$ | $3.19 \times 10^{-2}$ |
| md5sum | 0 | 0 | 0 |
| cxxfilt | $1.00 \times 10^{-8}$ | $6.82 \times 10^{-4}$ | $1.10 \times 10^{-3}$ |
| readpng | $2.00 \times 10^{-8}$ | $2.77 \times 10^{-3}$ | $3.00 \times 10^{-4}$ |
| bsdtar | $2.50 \times 10^{-7}$ | $8.19 \times 10^{-4}$ | $2.00 \times 10^{-3}$ |
| **Avg.** | $2.06 \times 10^{-6}$ | $5.38 \times 10^{-3}$ | $5.64 \times 10^{-2}$ |

TABLE X: False negative and $Recall_{diff}$ of HashMTI.

| **Subject** | **False Negative** | | **Difference of Recall** | |
|---|---|---|---|---|
| | Average | Variance | Average | Variance |
| who | 0.06 | 0.005 | 0 | 0 |
| avconv | 0.07 | 0.003 | $2.80 \times 10^{-3}$ | 0.000308 |
| ffmpeg | 0.09 | 0.003 | $9.10 \times 10^{-3}$ | 0.000711 |
| readelf | 0.10 | 0.010 | $1.79 \times 10^{-2}$ | 0.003903 |
| jhead | 0.10 | 0.009 | $2.13 \times 10^{-2}$ | 0.001621 |
| gif2png | 0.12 | 0.003 | $3.88 \times 10^{-2}$ | 0.001878 |
| base64 | 0.12 | 0.003 | $4.02 \times 10^{-2}$ | 0.003692 |
| pdftops | 0.13 | 0.009 | $7.38 \times 10^{-2}$ | 0.019262 |
| tiffinfo | 0.15 | 0.010 | $1.58 \times 10^{-2}$ | 0.000268 |
| tiff2pdf | 0.16 | 0.012 | $1.00 \times 10^{-4}$ | 0.000005 |
| uniq | 0.16 | 0.023 | 0 | 0 |
| tiff2bw | 0.18 | 0.016 | $7.90 \times 10^{-3}$ | 0.000910 |
| tcpdump | 0.18 | 0.020 | $2.74 \times 10^{-2}$ | 0.010290 |
| djpeg | 0.22 | 0.010 | $2.88 \times 10^{-2}$ | 0.008438 |
| infotocap | 0.26 | 0.014 | $6.95 \times 10^{-2}$ | 0.003388 |
| xmllint | 0.26 | 0.036 | $4.52 \times 10^{-2}$ | 0.004684 |
| pdfinfo | 0.26 | 0.009 | $2.68 \times 10^{-2}$ | 0.007069 |
| objdump | 0.27 | 0.050 | $4.10 \times 10^{-3}$ | 0.001154 |
| md5sum | 0.28 | 0.035 | $1.50 \times 10^{-2}$ | 0.005410 |
| cxxfilt* | 0.29 | 0.012 | – | – |
| readpng* | 0.32 | 0.034 | – | – |
| bsdtar | 0.38 | 0.061 | $4.76 \times 10^{-2}$ | 0.011887 |
| **Avg.** | 0.19 | 0.018 | $2.46 \times 10^{-2}$ | 0.004244 |

*Libdft cannot analyze programs that obtain input data from stdin.

the 9 subjects. On `djpeg`, there are on average 35% of taint reports that can be found only by the *DoubleMutation* strategy. On `jhead`, it contributed 99% of taint reports at most. Thus, it greatly mitigates the limitation of the hash record scheme.

From the results in Table VIII, we can positively answer **RQ3** that *DoubleMutation* is efficient. It helps detect more taint information while only introducing a reasonable number of additional executions.

## *E. Evaluating the Accuracy of HashMTI (RQ4)*

As aforementioned, HashMTI is more efficient and scalable than the original MTI. However, its hash functions may have collisions that could conceal the changes in variable values. Moreover, though the limitation of hash records is greatly mitigated by the *DoubleMutation* strategy, it still causes HashMTI to fail to detect some taint information. Below we evaluate the influence of these two aspects on the accuracy of HashMTI.

**Hash Collision.** Notably, the optimized Splitmix64 has no collisions in our experiments. For the optimized DJBX33A, Table IX lists its collision rates. In column 2, the proportion of taint reports that HashMTI fails to detect due to collisions is only on the order of $10^{-6}$. The maximum collision rate is $9.92 \times 10^{-6}$ which occurs on `xmllint`, and the average is $2.06 \times 10^{-6}$. Especially, no collisions occur on the 4 programs of the LAVA-M benchmark. Moreover, according to columns 3 and 4, on average only 0.54% of branches and 5.64% of test cases have had collisions, respectively. Therefore, the hash collisions are negligible and cause almost no losses in the accuracy. We employed the optimized DJBX33A as the default function of HashMTI and used it in all other experiments since it is simpler and faster than the optimized Splitmix64.

**False Negative.** The hash collisions and the limitation of the hash record scheme could affect the accuracy of HashMTI and thus leads to false negatives (compared with original MTI). Table X reports the average false negative rate of HashMTI and its variance on each subject. From the results, we can see the rates are reasonable in general (19% on average). Half of them are lower than 20%, and most of them (20 of the 22) are lower than 30%. In particular, on `who`, `avconv`, `ffmpeg`, `readelf` and `jhead`, HashMTI can detect more than 90% of the taint information found by OrigMTI. The small variances also indicate that the results are statistically significant.

**Recall Ratio.** From the perspective of replacing traditional taint analysis, we further evaluated the accuracy of HashMTI with a traditional DTA tool Libdft. By respectively comparing HashMTI and OrigMTI with Libdft, we computed the differences in their recall ratios of taint reports as follows.

$$Recall_{diff} = \frac{\left|Reports_{OrigMTI} \cap Reports_{Libdft}\right|}{\left|Reports_{Libdft}\right|} - \frac{\left|Reports_{HashMTI} \cap Reports_{Libdft}\right|}{\left|Reports_{Libdft}\right|} \tag{1}$$

where the $Reports_{Libdft}$ are the taint information reported by Libdft, represented by tuples of the input byte offset and tainted branch instruction.

Table X lists the $Recall_{diff}$ on each subject. The averages are all positive numbers due to the aforementioned false negatives of HashMTI. Nevertheless, they are at least one order of magnitude lower than the false negative rates, and the overall average is only $2.46 \times 10^{-2}$. Especially, HashMTI achieves the same recall ratio as OrigMTI on `who` and `uniq`. The small variances also indicate that the results are statistically significant. Therefore, HashMTI achieves similar accuracy as OrigMTI when compared with the traditional DTA tool Libdft.

**Similar Experiments as GREYONE.** Last but not least, we perform similar experiments as those in GREYONE [15]

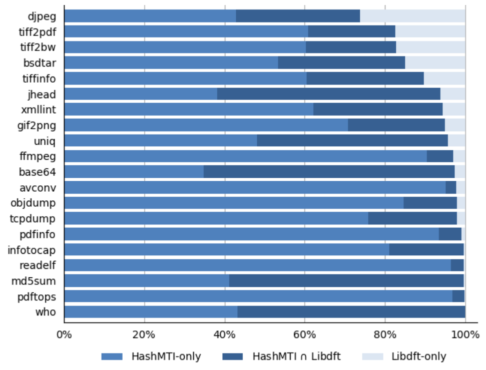


Fig. 5: Proportion of tainted branches reported by HashMTI-only, Libdft-only and both HashMTI and Libdft.

to evaluate the accuracy of HashMTI. We tested HashMTI and a DTA tool with the same subjects and input data (see Table III). Then we counted the number of tainted branches reported by them, respectively. Note that we use an out-of-the-box DTA tool Libdft [17] instead of the DFSan [40] used in GREYONE since the latter relies on our implementations as well.

Fig. 5 shows the proportion of tainted branches reported by HashMTI and Libdft. The experimental results of HashMTI are in agreement with those in GREYONE. We can see that HashMTI recalls most taint reports of Libdft while detecting more taint information than the DTA tool.

From the analysis of Table IX, X and Fig. 5, we can positively answer **RQ4** that HashMTI maintains a considerable degree of accuracy, especially from the perspective of replacing traditional DTA.

## VI. Threats to Validity

Since the baseline OrigMTI is implemented by ourselves, we perform similar experiments as those in previous works for comparisons. We are also eager to compare HashMTI with the MTI methods used in SLF and GREYONE after they release their code. Since they both face the challenges described in this paper, we believe that HashMTI can outperform them.

Besides, the performance improvement of HashMTI could be insignificant on simple programs. In this case, the overhead of runtime hash computations could even cause HashMTI to be slower than the original MTI. Nevertheless, our experiments showed that HashMTI can significantly speed up the taint inference on all of the 22 subjects. In particular, HashMTI scales well to large programs and thus is more practical.

## VII. Related Work

### A. Static Taint Analysis

Static taint analysis is the static counterpart of DTA. It does not need to actually run target applications and thus is widely used to analyze android [41] [42] and IoT [43] applications. It also has the advantage of computing conservative estimates of information flows within a program, whereas DTA can only identify flows that actually occur in observed execution paths [44]. However, the technique is imprecise. It produces many spurious results in the presence of constructs such as loops and aliases. Compared with static taint analysis, HashMTI is a dynamic experimental approach and thus is more sound and can detect information flows that static taint analysis cannot.

### B. Dynamic Taint Analysis

DTA tags sensitive input data with taint labels and propagates these labels to further variables that are computed based on the tagged ones. This technique is more precise than the static taint analysis since it is based on actual executions. Especially, some DTA tools [17] [40] use byte-level tagging granularity and thus can identify input bytes that influence branch conditions or program attack points [45].

However, DTA has heavy overhead for interpreting instructions and storing taint labels. It also suffers from under-taint issues caused by implicit data flows [13] and external function calls. In contrast to DTA, HashMTI is achieved by lightweight instrumentations. Moreover, it is an experimental approach that shows true counterfactual causality and thus soundness by construction [11].

### C. MTI-like Techniques Based on Control Flow Features

There are several works that have a similar idea as MTI, such as FairFuzz [46] and Profuzzer [47]. They systematically mutate each input byte as well but inferred the relationships between input and branch conditions based on changes in control flow features. However, the control flow features are coarse-grained indicators so that they fail to provide accurate taint information. They also spend too much time systematically mutating each input byte, which is not as effective as random strategies [20]. In contrast to these techniques, HashMTI maintains the accuracy of MTI while significantly improving its efficiency and scalability.

## VIII. Conclusion

In this paper, we propose an efficient and scalable MTI method, named HashMTI, which uses the hash of each variable's values to efficiently monitor their changes and further improves the performance of MTI. It is mainly made up of two parts: (1) the hash record scheme to fundamentally reduce the space complexity of original MTI and (2) the *DoubleMutation* strategy to mitigate the limitation of hash record and detect more taint information. The experimental results show that HashMTI reduces the space complexity of MTI to $O(1)$, achieves significant speedups due to better utilization of the CPU cache and maintains a considerable degree of accuracy.

Since the MTI is suitable for optimizing fuzzing and our work HashMTI further improves its efficiency, in the future, we plan to develop a taint-guided fuzzer based on HashMTI.

## Acknowledgment

We would like to thank anonymous reviewers for their valuable comments and helpful suggestions. This work was supported by the Natural Science Foundation of Hunan Province in China (2019JJ50729), the Natural Science Foundation of China (61902412, 61902416), and the Research Project of National University of Defense Technology (ZK20-17).

## REFERENCES

[1] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 116–127.

[2] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering." in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.

[3] D. Korczynski and H. Yin, "Capturing malware propagations with code injections and code-reuse attacks," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1691–1708.

[4] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 133–143.

[5] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 1–20.

[6] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 317–331.

[7] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing." in *NDSS*, vol. 17, 2017, pp. 1–14.

[8] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[9] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for overflows: A guided fuzzer to find buffer boundary violations," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 49–64.

[10] X. Mi, B. Wang, Y. Tang, P. Wang, and B. Yu, "SHFuzz: Selective Hybrid Fuzzing with Branch Scheduling Based on Binary Instrumentation," *Applied Sciences*, vol. 10, no. 16, p. 5449, 2020.

[11] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller, "Detecting information flow by mutating input data," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2017, pp. 263–273.

[12] D. Lewis, "Causation," *The journal of philosophy*, vol. 70, no. 17, pp. 556–567, 1974.

[13] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation." in *NDSS*, 2011.

[14] W. You, X. Liu, S. Ma, D. Perry, X. Zhang, and B. Liang, "SLF: Fuzzing without Valid Seed Inputs," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 712–723.

[15] S. Gan, C. Zhang, P. Chen, B. Zhao, X. Qin, D. Wu, and Z. Chen, "GREYONE: Data Flow Sensitive Fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2577–2594.

[16] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 110–121.

[17] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012, pp. 121–132.

[18] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[19] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966.

[20] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2307–2324.

[21] "libjpeg-turbo." [Online]. Available: https://libjpeg-turbo.org/

[22] M. Zalewski, "American fuzzy lop," 2014. [Online]. Available: http://lcamtuf.coredump.cx/afl

[23] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[24] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 679–696.

[25] C. Estébanez, Y. Saez, G. Recio, and P. Isasi, "Performance of the most common non-cryptographic hash functions," *Software: Practice and Experience*, vol. 44, no. 6, pp. 681–698, 2014.

[26] G. Cheng and Y. Yan, "Evaluation and design of non-cryptographic hash functions for network data stream algorithms," in *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE, 2017, pp. 239–244.

[27] Y. Saez, C. Estebanez, D. Quintana, and P. Isasi, "Evolutionary hash functions for specific domains," *Applied Soft Computing*, vol. 78, pp. 58–69, 2019.

[28] D. Grochol and L. Sekanina, "Evolutionary design of fast high-quality hash functions for network applications," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 2016, pp. 901–908.

[29] A. Partow, "General purpose hash function algorithms," 2013. [Online]. Available: http://www.partow.net/programming/hashfunctions/

[30] A. Appleby, "Murmurhash3," 2012. [Online]. Available: https://github.com/aappleby/smhasher/blob/master/src/MurmurHash3.cpp

[31] P. Hsieh, "The superfasthash function." [Online]. Available: http://www.azillionmonkeys.com/qed/hash.html

[32] "One At a Time Hash." [Online]. Available: https://en.wikipedia.org/wiki/Jenkins_hash_function

[33] "Splitmix64 hash." [Online]. Available: http://xorshift.di.unimi.it/splitmix64.c

[34] B. Jenkins, "Hash functions for hash table lookup," 1997. [Online]. Available: http://www.burtleburtle.net/bob/hash/doobs.html

[35] P. V. G. Fowler and L. C. Noll, "Fvn hash," 1991. [Online]. Available: http://www.isthe.com/chongo/tech/comp/fnv/

[36] U. Kargén and N. Shahmehri, "Turning programs against each other: High coverage fuzz-testing using binary-code mutation and dynamic slicing," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 782–792.

[37] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 2123–2138.

[38] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[39] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[40] "LLVM Dataflowsanitizer." [Online]. Available: https://clang.llvm.org/docs/DataFlowSanitizer.html

[41] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *SIGPLAN Not.*, vol. 49, no. 6, p. 259–269, Jun. 2014.

[42] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 106–117.

[43] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac, "Sensitive Information Tracking in Commodity IoT," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1687–1704.

[44] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 196–206.

[45] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 474–484.

[46] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd*

*ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 475–485.

[47] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, “ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery,” in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 769–786.