Fuzzing@Home: Distributed Fuzzing on Untrusted Heterogeneous Clients

Daehee Jang
Sungshin Women's University
South Korea
djang@sungshin.ac.kr

Ammar Askar
Georgia Institute of Technology
USA
aaskar@gatech.edu

Insu Yun
KAIST
South Korea
insuyun@kaist.ac.kr

Stephen Tong
Georgia Institute of Technology
USA
stong@gatech.edu

Yiqin Cai
Georgia Institute of Technology
USA
epp310@gatech.edu

Taesoo Kim
Georgia Institute of Technology
USA
taesoo@gatech.edu

ABSTRACT

Fuzzing is a practical technique to automatically find vulnerabilities in software. It is well-suited to running at scale with distributed computing platforms thanks to its parallelizability. Therefore, individual researchers and companies typically setup fuzzing platforms on multiple servers and run fuzzers in parallel. However, as such resources are private, they suffer from financial and physical limits. In this paper, we propose Fuzzing@Home; the first *public* collaborative fuzzing network, based on heterogeneous machines owned by potentially untrusted users. Using our system, multiple organizations (or individuals) can easily collaborate to fuzz a software of common interest in an efficient way. One can participate and earn economic benefits if the fuzzing network is tied to a bug-bounty program, or simply donate spare computing power as a volunteer.

If the network compensates collaborators, system fairness becomes an issue. In this light, we devise a system to make the fuzzing results verifiable and devise cheat detection techniques to ensure integrity and fairness in collaboration. In terms of performance, we devise a technique to effectively sync the global coverage state, hence minimizing the overhead for verifying computation results. Finally, to increase participation, Fuzzing@Home uses WebAssembly to run fuzzers inside the web browser engine, allowing anyone to instantly join a fuzzing network with a single click on their mobile phone, tablet, or any modern computing device. To evaluate our system, we bootstrapped Fuzzing@Home with 72 open-source projects and ran experimental fuzzing networks for 330 days with 826 collaborators as beta testers.

CCS CONCEPTS

• **Security and privacy** → **Software and application security**; • **Computing methodologies** → *Distributed computing methodologies.*

RAID 2022, October 26–28, 2022, Limassol, Cyprus

ACM ISBN 978-1-4503-9704-9/22/10…$15.00
https://doi.org/10.1145/3545948.3545971

ACM Reference Format:
Daehee Jang, Ammar Askar, Insu Yun, Stephen Tong, Yiqin Cai, and Taesoo Kim. 2022. Fuzzing@Home: Distributed Fuzzing on Untrusted Heterogeneous Clients. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3545948.3545971

1 INTRODUCTION

Fuzzing repeatedly searches a program's code paths using genetic algorithms. Because the repetition is parallelizable, increasing the rate of execution tends to yield more findings in a shorter period. To maximize the number of executions, fuzzers typically support parallel execution [55, 56], across multiple cores and distributed fuzzing infrastructure span multiple machines. Google's ClusterFuzz [24] project, for example, distributes fuzzing tasks among thousands of virtual machines [44], which collectively fuzz a vast amount of code. Presently, distributed fuzzing platforms are limited by the amount of computing power owned by a single entity.

In this paper, we present Fuzzing@Home— a collaborative *public* network for fuzzing. While existing distributed fuzzing projects [24] are built on top of trusted nodes, Fuzzing@Home aims to utilize public computation nodes owned by multiple entities, which can be potentially *hostile*. If the network participants are anonymous, they can potentially break the fairness/security of the system[1]. This is analogous to other collaborative distributed computing works that discuss fairness and cheating issues [15, 22].

Existing solutions to prevent cheating in collaborative computing rely on the fact that their computations are purely deterministic and mathematical (e.g., AES decryption). Therefore, a proof-of-work (PoW) model can effectively check if participating nodes are collaborating in a fair way. Unfortunately, computations in fuzzing targets are arbitrary (e.g., audio stream parsing, data sorting, string manipulation); hence it is impossible to apply existing proof-of-work.

To address this issue, Fuzzing@Home introduces Proof-of-Fuzzing-Work (PoFW), which is a variation of PoW tailored for distributed fuzzing. As in PoW, a high-level idea of PoFW is giving a network collaborator a challenge to prove if they are working as expected. This challenge is often based on brute-forcing cryptographic

[1]Results in fuzzing could be tied to money due to bug-bounty programs; thus raising fairness/cheating issues. Our threat model assumes clients are untrusted but servers are.

hashes (e.g., Bitcoin) in conventional systems. Unfortunately, such an approach is inapplicable in Fuzzing@Home because there are too many forms of computations in fuzzing; and some of them do not even produce any measurable output (e.g., `void` function).

As a solution, we devise a concept called an *execution hash*, which is a hash of the *code coverage map* of program execution. However, such an alternative needs to consider several non-trivial properties, such as coverage map saturation, non-determinism in binary execution, and so forth; which we investigate in this paper.

Fuzzing@Home also aims to be a large-scale distributed fuzzing platform that accepts heterogeneous clients as a computation resource. To this end, we use WebAssembly (WASM) to run fuzzers inside web browsers. We ported 30 open source projects from OSSFuzz to run with WASM. As WASM-fuzzers run inside a web browser, a single click/tap on a URL link allows any user to participate in Fuzzing@Home instantly. Additionally, to better de-duplicate computation across participants, we propose the idea of *global coverage synchronization* to increase the overall efficiency of the distributed system.

For evaluation, we bootstrap Fuzzing@Home in a private lab environment and also deploy it as an experimental real-world service to limited beta testers for 330 days. The beta service is based on 72 fuzzing pools (42 Linux based, 30 WASM based) powered by 826 collaborators. (77.3% users participated via their mobile phones/tablets using WASM). As a result, Fuzzing@Home found 37 bugs, which we reported to the project maintainers.

In summary, our key contributions are as follows:

- We design and implement the first public and collaborative fuzzing network, Fuzzing@Home.
- We solve security challenges stemming from using untrusted machines.
- We introduce techniques to minimize the performance cost of Fuzzing@Home's security measures.
- We use WebAssembly to include a wide range of heterogeneous fuzzing clients.
- We deploy Fuzzing@Home as an experimental beta service and gather live data for our study.

2 ASSUMPTIONS AND GOALS

2.1 Threat Model and Assumptions

Unlike previous networked fuzzing infrastructures [18, 19, 24, 33, 44], which assume that all nodes are trustworthy, we account for nodes that are dishonest or even malicious. Also, as our trusted computing base includes a centralized control server, malicious denial-of-service attempts are also threats that we must address. In our threat model, the control servers are trustworthy entities. This setup is well aligned with real bug-finding services such as bug bounty programs [5, 9, 23, 26, 58]; a security researcher, who reports a vulnerability, trusts the integrity of these programs.

2.2 Goals

Scalability. Fuzzing@Home's main goal is to build a large-scale public infrastructure for collaborative fuzzing. Thus, it is important to design all tasks to be independent and asynchronous, avoiding a single performance bottleneck in critical paths. To achieve this, Fuzzing@Home broadcasts the global state to participants and enables them to verify new findings locally. In particular, a node locally checks the novelty of its report (e.g., coverage) using its synchronized global view before submitting it to the control server for final validation. Thus, if another node has already reported the same discovery (thus registered as global knowledge), the current node discards its finding to avoid duplicating the control server's verification work. (§4.4).

Trustworthiness. Fuzzing@Home must assume network participants could be hostile. Specifically, we theorize two attack types: ① a greedy node that claims rewards without actually working (*goofing off*), ② a dishonest node that performs work but hides crashes and new corpus entries (*stashing*). Given that the attacker in our model physically owns the machine, our primary goal is to design the system to be less beneficial to malicious actors than to honest users.

Fairness. To share beneficial outcomes equitably, Fuzzing@Home needs to fairly evaluate a user's contribution to the network, considering multiple factors such as the difficulty of bug finding, the amount of contributed coverage information, etc. Unlike mathematical puzzles (e.g., PoW in cryptocurrencies), measuring and controlling the difficulty of fuzzing is fundamentally more challenging because of the underlying dynamics in finding new bugs. In general, the outputs from fuzzing (i.e., finding interesting new test cases or crashes) tend to plateau and become more challenging to find over time. However, the discovery of a new test case can also create a sudden, temporary drop in difficulty. This is because a new test case might pass some complicated constraints on an interesting code path and spark an influx of new crashes and test cases. To take the varying difficulty into account, Fuzzing@Home compensates new discovery with reward proportionate to the entire pool's computing effort consumed for the finding. As a result, if a crash was discovered with a small amount of fuzzing, its compensation is low; and vice versa (§3.5).

3 SYSTEM DESIGN

3.1 Infrastructure

Figure 1 depicts Fuzzing@Home' infrastructure, consisting of two main components, a trusted central authority referred to as the *control server* and group of *untrusted fuzzing nodes*. This section explains the roles of these components and their interaction in Fuzzing@Home.

Fuzzing Node. A fuzzing node is an individual entity that runs a fuzzer to provide computation power. Fuzzing nodes are grouped into fuzzing pools—a set of nodes targeting the same application. Using Docker or Web-Assembly, a unified runtime environment is used by all nodes and a select set of parameters (fuzzing seeds, coverage information, etc) are provided. The node then begins fuzzing the target application, starting with provided seeds and a hash-challenge given from the control server to prove the fuzzing work. Once a node finds a solution (*Proof-of-Fuzzing-Work*), control server can verify that the node correctly performed fuzzing as told. We call this request-response event a *fuzzing transaction* (i.e., a control server requests a fuzzing transaction to a participant node, and the participant responds to it).

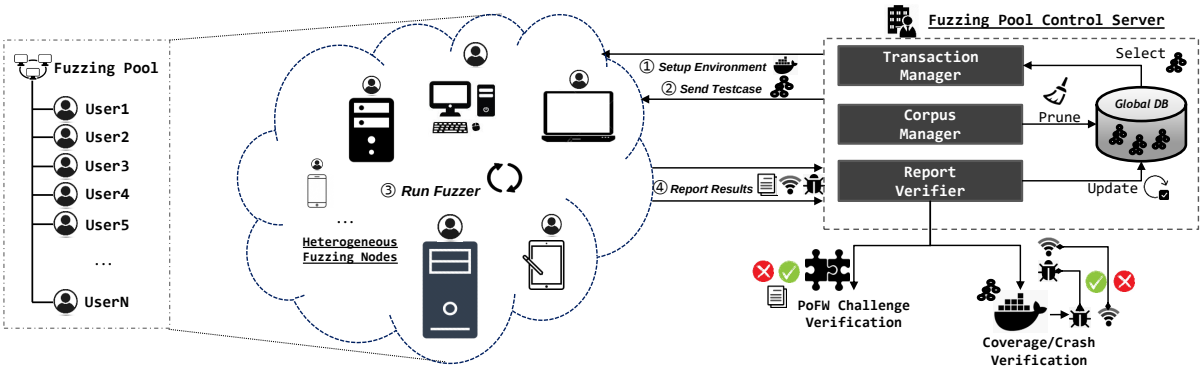

Figure 1: Overview of Fuzzing@Home infrastructure from a distributed fuzzing perspective. Fuzzing pools are orchestrated with a control server to distribute work and verify fuzzing results. *Transaction manager* handles seed selection and work load distribution. *Report verifier* validates results from fuzzing nodes and updates global corpus. *Corpus manager* minimizes fuzzing corpus and optimizes fuzzing seeds.

Control Server. The control server (trusted entity) orchestrates and controls all the fuzzing nodes within a single pool. This orchestration boils down to three key components:

- Distributing work load efficiently to avoid duplication (Transaction Manager).
- Verifying results and updating global information (Report Verifier).
- Cleaning up old/duplicated results (Corpus Manager).

To begin a fuzzing transaction, the control server selects an unexplored range of seed numbers for input space, initial inputs for mutation, and challenge execution-hash as a puzzle; a seed number (not a fuzzing seed corpus) is an integer that affects the input mutation algorithm. To solve the puzzle most cheaply, a node honestly executes the given fuzzer with the given parameters, then finds a test case to serve as a proof-of-fuzzing work. While solving the puzzle, a node might additionally discover interesting test cases that expand previous code coverage or that cause new crashes. If a test case expands the code coverage globally for the entire fuzzing pool perspective, the control server accepts it (and optionally compensates the user) as a discovery and adds it to the global test case storage.

3.2 Proof-of-Fuzzing-Work (PoFW)

Fuzzing@Home aims to constantly provide baseline reward points to fuzzing nodes in return for invested computing power, regardless of their discovery results. This acts as a motivator for users to join Fuzzing@Home even if their individual chances of finding a crash is limited. In combination with our threat model of accounting for malicious nodes, this reward mechanism opens up a potential problem. We have to verify that nodes are actually running the given fuzzer and not claiming the associated reward while skipping the required computation (referred to as *goofing attack*).

One way to prevent such goofing attack is using Proof-of-Work (PoW). However, because computation in fuzzing target could be arbitrary; conventional PoW mechanisms such as crypto-operation challenge are incompatible. To solve this issue, Fuzzing@Home introduces Proof-of-Fuzzing-Work (PoFW), which is similar to existing Proof-of-Work (PoW) mechanisms [28, 29, 36] used in cryptocurrencies but applicable to arbitrary computation.

The idea of PoFW follows these high-level steps: ① the control server randomly picks a fuzzing seed number and an initial input from corpus. ② the control server generates *execution hash* by running the fuzzer with these settings; the execution hash is a *cryptographic hash of code coverage information (e.g., data structure such as coverage map, inline counter array) achieved by running the fuzzer.* ③ the control server challenges a node to find the corresponding seed number that results the same hash as a puzzle with some hints to adjust difficulty, and finally ④ the node exhaustively searches the answer among the given range of seed numbers. If the node finds the answer, the control server verifies its correctness within `O(1)` time/memory complexity. It is possible for a node to try to skip some of the search space and still luckily find the answer in a smaller time frame. If this strategy works consistently, the challenge-solving will become adjusted accordingly, but if it doesn't then the attacker will have to do an exhaustive search. These steps are depicted in Figure 2.

One question in this design is the uniformity of these execution hashes. As execution hash is based on code coverage, the cryptographical properties of hashes (e.g., low collision rate) becomes questionable. In the worst case, say a program with two branches, PoFW mechanism would be impractical because an attacker could forge the answer and still be validated as legit node. To investigate this issue, we conduct experiments based on our PoFW implementation and assess the practicality in §5. Our experiment indicates that the majority of popular open-source applications (e.g., libpcap, zlib, etc) have sufficient entropy to approximate its code coverage information as a proper PoFW challenge for our system.

3.3 Global Coverage Synchronization

In addition to PoFW mechanism, Fuzzing@Home also validates discovery report for finding new code paths and crashes. Unlike PoFW reports, the control server cannot anticipate how many discoveries nodes will report in a given time. From the development of our prototype, we observed that fuzzing nodes in the early phase often report duplicate findings in bursts, yet they seem unique from each node's perspective. For example, two nodes might report a test case that results in new code coverage from their *local* perspective, when in reality, it results in the same coverage in the *global*

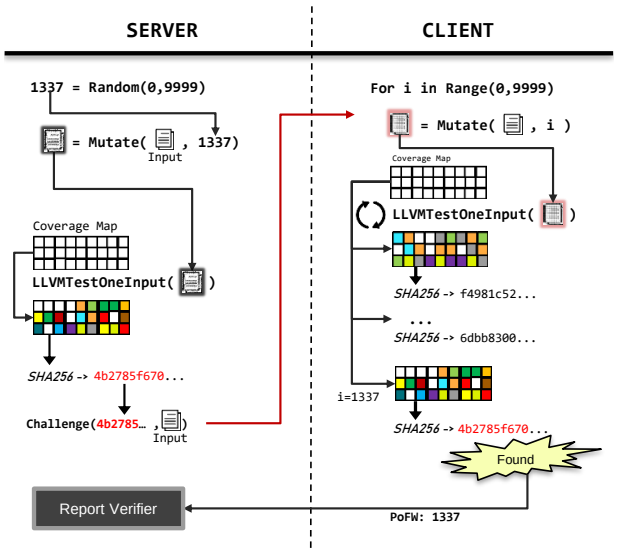

Figure 2: Proof-of-Fuzzing-Work challenge and response. Server randomly picks a number for the input mutation engine, then hashes a code coverage map running the mutated input. The server challenges the client to find the number that yields the same result. The server can adjust the difficulty by telling the client the range of the seed number.

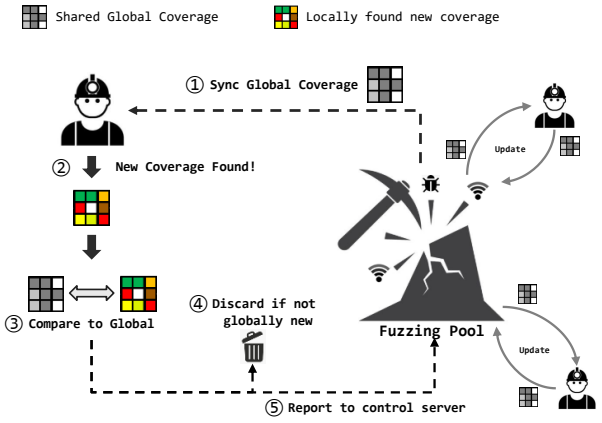

Figure 3: Global coverage synchronization in FUZZING@HOME. For each transaction, the client downloads the pool's global maximum coverage information. Every time a node locally discovers a new coverage expanding input, it can locally verify its finding against the pool's. As a result, a significant amount of unnecessary verification work is avoided.

scope. Thus, overly frequent/massive duplicate reports can happen; inducing a performance bottleneck in control server.

To reduce the control server's verification overhead for duplicate discoveries, FUZZING@HOME synchronizes global coverage data (e.g., entire coverage history merged/accumulated in the pool) with participating fuzzing nodes. This allows each fuzzing node to determine locally whether a newly found discovery is worth reporting from a global perspective. In other words, promptly syncing global coverage information across fuzzing nodes avoids duplicating reports and helps FUZZING@HOME to better scale, as depicted in Figure 3. Similarly, we also synchronize fingerprints of reported/verified crashes to allow client nodes to perform crash de-duplication on their end.

3.4 Stashing

A node could participate in FUZZING@HOME, gaining PoFW rewards through honest computing work, while withholding any discovered bugs. The attacker could then potentially try to weaponize or market these bugs themselves (e.g., claim a bug bounty). This attack is subtle and difficult to detect. One fundamental approach to stop stashing is to use a hardware-based trusted execution environment (TEE) to enforce the intended execution (e.g., reporting every crash). Unfortunately, hardware-based TEEs often require a specific processor model (e.g., Intel Skylake for SGX) and introduce significant deployment cost [43] to port fuzzers.

Another method is to detect stashing from a node's reporting behavior; if FUZZING@HOME knows that a particular transaction should crash a program, it can check a node's honesty by observing whether it reports the expected crash. For this purpose, we might inject fake bugs that crash an application; however, existing bug injection techniques [14, 42] have signatures for injected bugs, allowing an attacker to recognize them. For example, LAVA [14], one of the most famous bug injection tools, uses a specialized function, `lava_get()`, for introducing bugs. Moreover, with access to the original application version (e.g., open-source software), any bug injection technique is pointless; an attacker can distinguish fake bugs via differential testing.

Although stashing is hard to prevent systematically, we can discourages stashing economically with a simple first-come-first-serve compensation policy. We observe that because a nodes' maximum code coverage tends to grow together as they share seed corpus; once one node discovers a crash, another node that shares the corpus is likely to find the same crash soon. We note that this does not imply the whole concurrent fuzzing endeavor is redundant. In FUZZING@HOME, all nodes independently struggle to find new code paths; but, as they share previously discovered paths via accumulated seeds, global coverage information; the redundancy only applies for finding new path (i.e., all miners mining new block in block chain; versus, re-constructing entire block-chain), not the old paths. This tendency is discussed in a recent work on the empirical laws of fuzzing [6]. Crash discovery logs (e.g., `clamav`) in our evaluation data set also support this observation Table 6. In conclusion, if an honest node participating in the network (sharing seeds) suddenly attempts to withhold a discovery while collaborating with others, the discovery result will likely be outdated soon and other collaborator will be compensated. One might worry our global state synchronization, which reduces duplicate discovery, contradicts this claim. To clarify, we emphasize that our global coverage synchronization suppresses nodes from *reporting* the duplicate discovery, not *finding* them in the first place.

3.5 Rewards

If FUZZING@HOME is tied to a bug-bounty program, rewarding policy matters a lot. The experimental beta platform for FUZZING@HOME implements rewards with virtual points based on the amount of contributions in three folds: (i) solving PoFW

puzzles, (ii) reporting globally new code coverage, and (iii) reporting previously unseen crashes. In the following, we discuss Fuzzing@Home's reward system.

PoFW-based reward. A simple way to distribute rewards in Fuzzing@Home is based on PoFW (see §3.2). If a participant reports correct PoFW to a control server, a reward is given accordingly. Rewards based on PoFW will equally and steadily distribute benefits to participants, and the reward amount is easily calculable based on the computing power (e.g., 10 PoFWs per hour).

Discovery-based reward. Fuzzing@Home can also provide rewards for submitting a test case that leads to the pool's coverage expanding or the program crashing. If the participants actively seek to discover a promising test case on their own, on top of the given system (given fuzzer, seed inputs, etc); Fuzzing@Home can become a competitive fuzzing system, which motivates users to innovate on fuzzing by compensating them for contribution.

For crashes, it is vital to de-duplicate them without false positives to prevent malicious users from unfairly gaining rewards. Currently, Fuzzing@Home uses AddressSanitizer reports for de-duplication, similar to ClusterFuzz [24] but with a stricter configuration focusing on reducing false positives (i.e., different crashes might be considered the same, but not vice versa). We believe that more advanced techniques for crash de-duplication could be applied [34, 52, 54] but we leave this as future work.

To determine an appropriate reward amount, Fuzzing@Home uses a dynamic difficulty policy. Similar to conventional lottery systems like the Powerball [31], Fuzzing@Home initially sets a relatively small amount as a reward. The amount gradually increases over time until there is a winner (a new coverage expanding input or crash in Fuzzing@Home). If a node claims a discovery, Fuzzing@Home resets the amount to the initial value. As a result, easy-to-find discoveries yield small rewards and hard-to-find findings provide a higher premium.

4 IMPLEMENTATION

4.1 Fuzzing Node

We modified AFL 2.52b and libFuzzer from LLVM-11 to support our system features such as PoFW. In our deployment and evaluation, we mainly use libFuzzer because it has better performance and more diverse fuzzing capabilities compared to AFL. For Linux users, we used Docker to provide pre-built binaries and consistent environments for fuzzing. This is important because Fuzzing@Home's PoFW hash relies on a program's coverage map, which is highly dependent on the fuzzing binaries.

AFL. We modified AFL to 1) support PoFW hash generation, 2) synchronize coverage with a control server, and 3) reproduce fuzzing for verification. In particular, Fuzzing@Home hashes AFL's edge-coverage bitmap during fuzzing to generate the PoFW hash for a challenge. Since AFL uses a single bitmap data structure to manages its coverage, we share this map with the control server. This map has a fixed size of 64Kb, which represents the edge coverage of the execution. To produce a consistent PoFW hash for verification, fuzzing needs to be deterministic among multiple instances based on the given initial test case and a random seed. Thus, we modified AFL's mutation algorithm to use our seed (given by the control server), instead of seeding its RNG through an external source, e.g., `/dev/urandom`. Finally, we modify the fuzzer to take global coverage information into its memory. For AFL, such synchronization is achieved by substituting the 64Kb coverage map.

libFuzzer. Similar to AFL, we modified libFuzzer to support the aforementioned features. However, libFuzzer has a more complex notion of coverage measurement. Unlike AFL, which uses code coverage as its only coverage measure, libFuzzer has several coverage concepts, which are called features, to improve its mutations. For example, in LLVM-11, libFuzzer has a Table of Recently Compared Value (TORC) which is a dictionary data structure that tracks operand values of compare instructions to satisfy multi-byte comparisons (targeting magic values). Currently, Fuzzing@Home's global coverage synchronization only considers libFuzzer inline counter arrays (similar to coverage map in AFL) that quantify the edge coverage information.

WebAssembly Support. Initially, we did not support WebAssembly (WASM)-based fuzzers. However, user feedback from our preliminary experiment phase motivated us to support heterogeneous computing environments with easy setup. WASM is a great fit for this purpose as it runs inside web browsers. By simply navigating to a page, users can utilize their mobile phones, tablet, desktop and other computing devices regardless of their operating system (Windows, Mac, or Linux). To implement the WASM-based fuzzing pool, we use emscripten compiler with `SanitizerCoverage` in the build process, which instruments a binary with code coverage support. Then, the binary is linked with our modified and pre-compiled `libFuzzer.a` library because emscripten currently lacks libFuzzer support as part of compiler runtime [16]. After compilation, we use WebAssembly APIs to implement Fuzzing@Home protocols and other features such as global coverage synchronization. For ASAN [45] instrumentation, we maintain versions of the software w/ and w/o ASAN for efficiency. If the client machine has sufficient memory (e.g., desktop computer) we provide the ASAN instrumented version, otherwise not (e.g., mobile). To encourage user participation, we also put effort towards a friendly web interface. In particular, we use web-worker threads for real-time user interaction and allow users to manually mutate the libfuzzer test cases with their mouse and keyboard when the fuzzer is running. As humans can sometimes instinctively pick up on valid/invalid data patterns, they can potentially help solve constraints that computers cannot quickly solve through its algorithmic input generation and mutation. This feature is a graphical interface implementation akin to prior human-assisted fuzzing work introduced by HaCRS [48].

4.2 Control Server

To support as many nodes as possible, the control server is implemented using asynchronous frameworks — `aiohttp` [3] with communication over REST APIs.

Verification. For PoFW verification, the control server waits for an answer from a client. If the client does not answer the challenge in a configured time period, the challenge will be discarded. If an answer arrives in time, the server checks its validity and compensates the participant according to the policy, which can be adjusted based on the usage of Fuzzing@Home. To verify coverage expanding and crashing inputs, the control server runs the fuzzing target with the given input using the same isolated environment as the

node. The control server then verifies the coverage expansion using its global coverage information and performs de-duplication for crashes. Unlike PoFW reports, the discovery rate for coverage and crashes drops significantly as new corpus accumulates (see §5.1). However, because this verification introduces a large overhead due to the execution cost of running the application, Fuzzing@Home limits the number of reports per transaction to mitigate a potential denial of service attack.

Corpus Pruning. The control server also needs to supply starting inputs to each node based on the global corpus in an efficient way to avoid running duplicated test cases. In the short-term, duplication can be avoided by randomly and evenly splitting the global corpus to distribute to nodes. However, in the long-term, proper pruning of the corpus is vital to reduce duplicates. As coverage expands, the probability that a new test case is a superset of an older one increases. In this case, mutating the old subset test case results in many duplicates covered by the newer test, resulting in wasted fuzzing effort. The control server implements corpus pruning with `afl-cmin` for AFL-based fuzzing pools and `-merge` option for libFuzzer based ones. The timing of corpus pruning is dynamic based on the number of coverage reports and total corpus size.

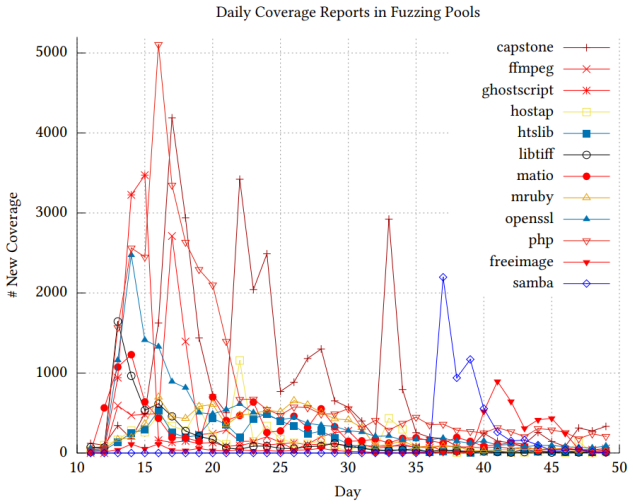

Figure 4: Daily coverage discovery trend in Fuzzing@Home since the public launch. The graph suggests that difficulty of finding new code coverage exponentially increases over time. Due to lack of space, this is a subset of the entire data that shows the overall tendency (full data shown in appendix).

4.3 Deployment

We deployed our implementation of Fuzzing@Home as a real-world service. [2] At the time of writing, Fuzzing@Home has been running for 330 days with 72 fuzzing pools and 826 unique users. We chose 72 open source projects part of Google's OSSFuzz [44] project to aid with comparing to an existing distributed fuzzing project. Each project has a fuzzing pool with a dedicated control server. We used 42 projects based on Docker on Linux and 30 based on WASM. For non-OSSFuzz projects, we collaborated with Bitcoin maintainers and released a fuzzing pool that contains 142 fuzzing harnesses for fuzzing the Bitcoin Core project. Any open-source project can be integrated with Fuzzing@Home' build system by replacing the underlying compiler. During our deployment, we computed 5,908 G fuzzing executions (with a peak rate of 1.8 M exec/sec) and accumulated over 500 K unique test corpora (# of input files). As a result, we found 37 unique bugs as summarized in Table 1.

[2] for anonymity, we omit information to access this service.

Project	# Unique Bugs	Description
Apache Arrow	1	null pointer dereference
ClamAV	2	heap-read-buffer-overflow null pointer dereference
FreeImage	5	stack-write-buffer-overflow out-of-memory allocation-size-too-big heap-write-buffer-overflow global-read-buffer-overflow
Capstone	1	global-read-buffer-overflow
htslib	1	out-of-memory
libtiff	1	out-of-memory
matio	21	calloc-overflow allocation-size-too-big out-of-memory SEGV on unknown address (9) stack-write-buffer-overflow heap-read-buffer-overflow (5) heap-write-buffer-overflow memcpy-param-overlap floating point exception
Samba	1	heap-read-bufferoverflow
Xvid	1	heap-read-bufferoverflow
mruby	1	out-of-memory
stb	1	heap-read-buffer-overflow
quickjs	1	heap-read-buffer-overflow
Total	37	unique bugs found

Table 1: Unique bugs found during Fuzzing@Home's public deployment. The numbers in the parenthesis in `matio`'s row is the unique bug count based on manual analysis. The entire history of bug discovery and analysis can be found in Table 6.

Figure 4 visualizes the tendency of code coverage accumulation over time in fuzzing pools. From the data, we can see coverage discovery peaks dramatically in the short period after deployment then tanks in a few days as code paths saturate. This significantly reduces the control server overhead in time. During the service run, we found 37 unique bugs and reported all found bugs to vendors. The number of bugs was originally 191 based on existing de-duplication technique. Due to limitations in these techniques, after manual root cause analysis and collaboration with the vendors, we arrived at the unique bug number. Some of the bugs found during Fuzzing@Home' deployment were also simultaneously reported by ClusterFuzz (through OSSFuzz). As mentioned before, our fuzzing targets are a subset of OSSFuzz's (except Bitcoin), which has been fuzzed for years. At this point, Fuzzing@Home is still far from ClusterFuzz in terms of scale (using 26,000 cores). However, our preliminary deployment shows that Fuzzing@Home is effective at discovering bugs as an open fuzzing platform and may outperform ClusterFuzz, given increased public participation.

4.4 Load Balancing for Scalability

Fuzzing@Home's control server adjusts its transaction load by dynamically measuring the overhead for each fuzzing transaction. When a client node requests a fuzzing transaction, the control server not only creates a PoFW challenge but also anticipates an amount of computation time to solve the puzzle. The amount of time is dependant to computing power of existing nodes. Initially, a control server cannot accurately predict the expected time to solve the puzzle; thus it empirically assigns the pre-defined size of the transaction. Over time, the control server measures the response time of clients and adaptively regulates the amount of work. For example, if the client solves a puzzle too fast, the next transaction size doubles up (exponential back off). Likewise, if the client solves a puzzle too slowly, the next transaction size will be reduced in half. In this way, the control server regulates expected time interval between PoFW reports. In our prototype, this interval is set to 10 minutes. This can be changed based on the control server's computation power and the number of participating clients in the pool.

Additionally, to handle malicious clients DoS-ing the control's verification load with a massive amount of fake reports, the control server configures minimum/maximum timeout range and filters out extreme cases. Reports outside of such expected range are considered malicious and discarded to protect the control server from denial-of-service attacks.

5 EVALUATION

We evaluate Fuzzing@Home using our private computing cluster and real-world logs collected from Fuzzing@Home users. Our private computer cluster for evaluation consisted of 18 identical servers with 24-core AMD Ryzen CPUs, 32GB RAM, and 512GB SSDs. In this section, we attempt to answer the following questions:

- How scalable is Fuzzing@Home with respect to the number of nodes and global coverage synchronization? (§5.1)
- How much overhead does Fuzzing@Home incur to run on top of untrusted nodes? (§5.2)
- Is it okay to use coverage maps for PoFW? (§5.3)
- How effective is Fuzzing@Home's cheat prevention? (§5.4)
- What are the costs of WASM-based fuzzing? (§5.5)

5.1 Scalability

To evaluate Fuzzing@Home' scalability, we constructed four private fuzzing pools. We chose two big projects (`capstone` and `FreeImage`), and two small projects (`libmpeg2` and `WavPack`) based on the size of their codebases, as shown in Table 5. Then, we measured the overall execution speed of each fuzzing pool while gradually increasing the number of nodes up to four hundred, which is the maximum possible node number on our available hardware.

Figure 5 shows our results; Fuzzing@Home eventually scales for every project that we tested (i.e., after 48 hours). However, if we focus on the initial phase of the fuzzing pool (before coverage saturation), we observe that overall performance is not proportional to number of nodes in `capstone` and `FreeImage`. This is due to a large amount of *coverage discovery reports*. In our evaluation, both `capstone` and `FreeImage` took more than 24 hours before code coverage saturates. Therefore, during the initial 6 hours, the control servers for both `capstone` and `FreeImage` are under a colossal coverage verification load, thus slowing down the overall fuzzing. For `WavPack` and `libmpeg2`, coverage saturates within an hour, thus quickly relieving the control server from the verification load. This evaluation shows the importance of reducing duplicated coverage reports in Fuzzing@Home.

Project	1st	2nd	3rd	Project	1st	2nd	3rd
arrow	7.3%	6.6%	5.9%	lame	1.6%	1.0%	0.1%
binutils	21.5%	14.7%	13.3%	libmpeg2	0.3%	0.2%	0.1%
capstone	0.8%	0.4%	0.1%	libpcap	37.1%	5.6%	2.2%
c-ares	33.8%	5.6%	1.8%	libpng-proto	11.6%	0.9%	0.5%
eigen	32.4%	18.6%	14.6%	libtiff	10.0%	3.6%	2.8%
ffmpeg	0.6%	0.2%	0.1%	libzip	1.7%	0.8%	0.4%
flac	6.2%	5.4%	3.0%	lodepng	26.8%	23.8%	17.3%
freeimage	1.4%	1.2%	1.0%	matio	25.5%	8.1%	7.0%
gfwx	32.6%	5.4%	3.4%	mruby	1.5%	0.2%	0.1%
giflib	31.4%	9.8%	2.8%	ntp	26.7%	6.4%	5.6%
htslib	2.1%	0.3%	0.1%	php	18.3%	2.9%	0.3%
jansson	4.1%	4.0%	3.2%	wavpack	2.2%	0.1%	0.1%
kcodec	0.6%	0.4%	0.1%	zlib	0.2%	0.1%	0.1%

1st: Highest percentage of duplicated hashes
2nd: 2nd Highest percentage of duplicated hashes
3rd: 3rd Highest percentage of duplicated hashes

Table 2: Three highest hash-duplication-ratios among 1M executions. Inputs are auto-generated by libfuzzer mutation from empty corpus. If the input mutation is too small, the program will take exact same code path; producing same coverage map.

We claim Fuzzing@Home can achieve scalability because main loads are either 1) linearly proportionate to number of clients or 2) high/unpredictable but exponentially decrease over time (Figure 6). The number of transaction scheduling tasks is linearly proportional to the number of nodes, and its overhead can be very minimal because control server can adjust (increase) the difficulty of PoFW challenge. Verifying coverage and crash reports from nodes incurs high overhead because Fuzzing@Home needs to re-run a target application under isolated docker environment to securely validate each claims. However, as shown in Figure 6, the number of coverage discovery reports decreases exponentially over time, making the long-term overhead insignificant. This drop comes from a tendency that the discovery of new coverage becomes more difficult as time goes on [6]. As seen in the figure, global coverage synchronization in Fuzzing@Home further reduces this overhead by forcing nodes to validate their claim locally before submitting it to the control server.

5.2 Overhead for using Untrusted Node

To measure the overhead of Fuzzing@Home' security-centric design and features, we use ClusterFuzz (Google's distributed fuzzing project) as a baseline for comparison. We ported ClusterFuzz 1.8.0 to run locally on our machines without interacting with Google Cloud Storage (GCS). To replace GCS, we patched `gsutil` commands and modified them to use SFTP. ClusterFuzz provides two setup environments: production setup and local setup. In the production setup, users have all the functionalities provided by ClusterFuzz, with the help of Google Cloud services like Cloud Datastore, Google Cloud Storage, BigQuery. In the local setup, users do not have all the functionalities provided by ClusterFuzz. Datastore is running locally. Google Cloud Storage is replaced by the local file system. BigQuery

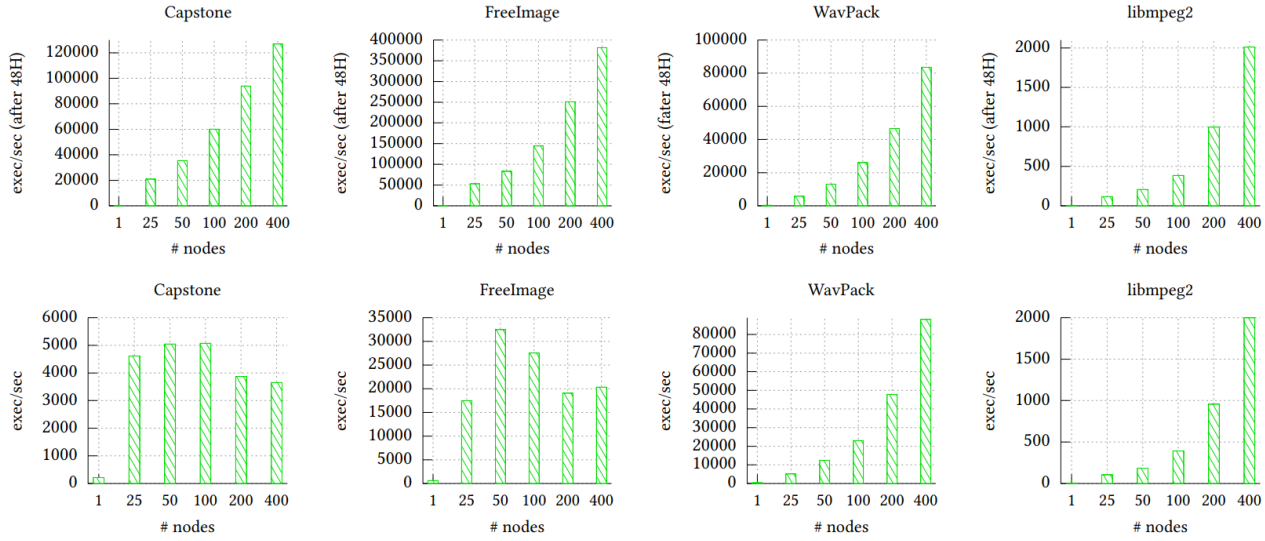

Figure 5: Fuzzing@Home scalability evaluation result. The upper row are exec/sec after running a fuzzing pool for more than 48 hours (to measure server stress after the coverage is saturated) while the lower row is average exec/sec speed in the initial 6 hours. (to measure control server's verification load during coverage saturation phase). `capstone` and `FreeImage` take more than 24 hours to saturate their coverage, while `WavPack` and `libmpeg2` take less than an hour. Coverage reports tend to drop off steeply once a project has been fuzzed for a few days.

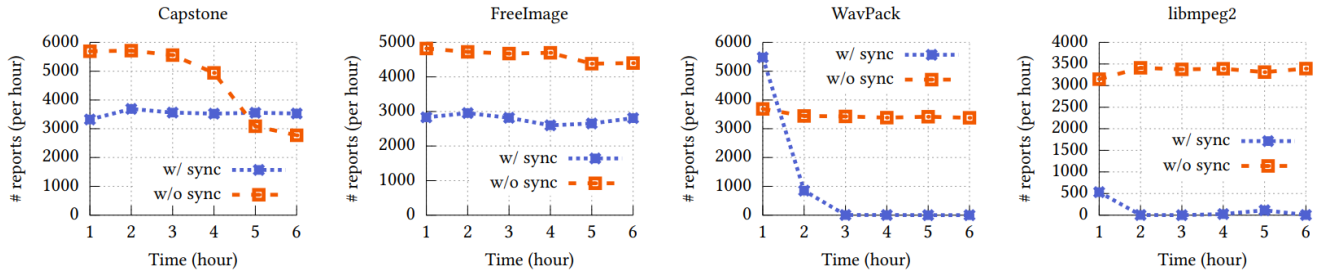

Figure 6: Fuzzing@Home coverage report frequency. With a new pool, we measured the number of coverage discovery reports with and without global coverage sync. The results indicate that global coverage sync substantially decreases the number of duplicate reports.

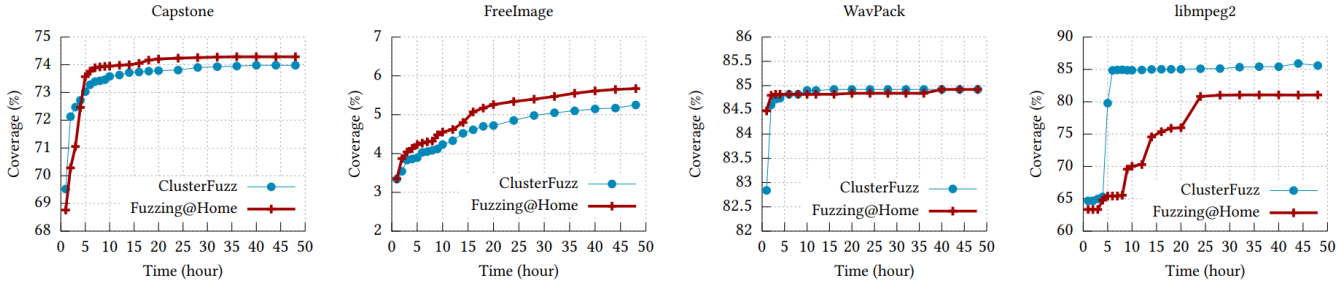

Figure 7: Coverage expansion rate for ClusterFuzz and Fuzzing@Home with identical initial corpus, hardware, and software environments.

related components are disabled. The server and bots run on the same machine. We use neither but modified the source code based on the local setup using version v1.8.0 with the goal of running ClusterFuzz on our cluster. To enable Datastore, `socat` is used to transfer local port messages with the outside. To replace Google Cloud Storage, all the file transmission between the server and the bots goes through SFTP instead. Figure 8 contains the ClusterFuzz parameters we used in our evaluation.

After porting, we ran ClusterFuzz and Fuzzing@Home with the same computing power (a `100` node cluster). We fuzzed the same applications as before (i.e., `capstone`, `libmpeg2`, `WavPack`, `FreeImage`) with Fuzzing@Home and ClusterFuzz for 48 hours and measured the difference in their overall code coverage expansion rate. Both systems use the same version of *libFuzzer* for their fuzzing engine, identical initial seed corpus, and runtime environment.

Project	# execution	Project	# execution
arrow	63K	lame	16K
binutils	125K	libmpeg2	14K
capstone	54K	libpcap	387K
c-ares	unseen	libpng-proto	492K
eigen	unseen	libtiff	318K
ffmpeg	233K	libzip	404K
flac	unseen	lodepng	unseen
freeimage	69K	matio	341K
gfwx	516K	mruby	23K
giflib	582K	ntp	unseen
htslib	462K	php	93K
jansson	unseen	wavpack	65K
kcodecs	7K	zlib	120K

 # execution: Number of executions until first hash deviation is observed.
unseen: Deviation not observed within 1M executions.

Table 3: Due to non-determinism, a program can yield different coverage map even with the same input. This form of non-determinism generally occurs due to asynchronous interrupts. If a program inherently depends on randomness (e.g., /dev/urandom), it must be patched.

Figure 7 shows the results. In Fuzzing@Home, due to the verification overhead, the executions per second of individual nodes are lower than that of ClusterFuzz. However, coverage synchronization between nodes earns more efficiency over ClusterFuzz. Overall, the results indicate there is no significant difference. What we see is that Fuzzing@Home outperforms ClusterFuzz in `FreeImage`, `capstone`, but falls behind in the case of `WavPack`, `libmpeg2`. Multiple factors can explain these results. One such factor could be that `FreeImage` and `capstone` are relatively large code-bases, thus there are a huge number of paths for new coverage, which are efficiently de-duplicated in Fuzzing@Home.

Parameter	Values
Distribution	`-fork=20`
Templates	`libfuzzer, engine_asan, prune`
Pruning Hour	12 hours
Environment Variables	MAX_TESTCASES=1, FORK_STRATEGY=1
Task Lease Time	1 hour (1 day default)
Corpus Pruning Timeout	1 hour (22 hours default)
OS Version	Ubuntu 16.04

Figure 8: ClusterFuzz Parameters used in our evaluation.

5.3 Collision Rate in Coverage Map

Duplicated Coverage Maps. As Fuzzing@Home utilizes a coverage map as an approximation of hash for a proof-of-work mechanism, we evaluate if this is a practical approach. Ideally, PoFW requires a unique execution hash for a unique input data. However, if a program is small, multiple inputs can trigger the same code path; thus yielding the same execution hash, which is a collision. To measure this collision rate, we randomly picked libfuzzer stubs in 26 open source software in OSSFuzz and gathered 1 million execution hashes from different inputs. Among the million hashes, we measured the percentage of duplicated ones and summarized the highest three cases in Table 2.

From the evaluation result, we observe a quite high collision rate (over 20%) in `binutils, c-ares, eigen, gfwx, giflib, libpcap, lodepng, matio, ntp`. We conducted manual analysis for such cases and found out reasons for such a high collision rate is (i) an input has a complicated format, (ii) we did not have an initial seed corpus, and (iii) a program is too small. For instance, `binutils (stub: fuzz_disassemble)` and `libpcap (stub: pcap_compile)` show a high collision rate because they were syntax parsers (opcode parsing, pcap language parsing); thus multiple malformed input take the same error path. In case of `matio, ntp`, we started fuzzing from empty seed corpus; thus there are trivial malformed immature inputs. `c-ares, eigen, gfwx, giflib, lodepng` had small libfuzzer stub code base (hundreds to thousands of LoC). For `matio, ntp, binutils, libpcap` collision rates will decrease over time as seed corpus gets mature after multiple fuzzing transactions.

The high collision rate indicates that a participant could fake a PoFW without executing the fuzzer. Our evaluation suggests that for OSSFuzz targets, a malicious participant might construct a fake PoFW answer and validated one out of ten trials in average. However, if the system administrator imposes even a small penalty for a wrong PoFW answer, goofing attacker will eventually earn negative rewards, therefore losing its advantage.

Non-determinism. If Fuzzing@Home penalizes participants for wrong PoFW responses, we must consider the case where an honest user unintentionally returns a wrong answer due to a program's non-deterministic behaviors. PoFW's hash calculation is based on a program's code coverage; thus ideally we need the path taken by a program's execution to be identical with repeated inputs under the same environment. Unfortunately, this is not always true, especially with larger and more complex programs. For example, if a system call gets interrupted with `EINTR`, a program could go down another path to retry it. In this case, the code coverage and, consequently, the execution hash will change.

To measure the extent of non-determinism in execution hash, we repeat fuzzing execution with exact same input under same conditions and gather results. Table 3 summarizes coverage map deviation rates for each target application. We iterated executions in 1K unit until we find a deviating execution hash. We put `unseen` in the graph in case we could not find deviation after 1M executions. The evaluation result indicates that the probability of unintentionally penalizing an honest user's PoFW report should be very small (less than 0.01% in most cases). If Fuzzing@Home requires higher PoFW accuracy, additional work would be required. We note, changing non-deterministic execution to be deterministic is an on-going research topic in general [13, 30, 35, 49].

5.4 System Fairness

Goofing Attack. To evaluate how effectively Fuzzing@Home can detect goofing, we synthesize a goofing attacker that fakes the PoFW answer based on the highest duplicated coverage map information as in Table 2. In our evaluation setup, a control server rewards user 1 point for each correct PoFW answer. If the participant is honest, the expected reward amount should be linearly proportionate to the number of solved PoFW challenges in any fuzzing pool. We synthesized cheaters in four fuzzing pools (`arrow`,

Project/Fuzzer	N-A	W-A	N-NA	W-NA
bzip2/bzip2_compress_target	2,004	270	2,501	2,598
bzip2/bzip2_decompress_target	17,597	2,909	26,402	21,146
guetzli/guetzli_fuzzer	21,776	2,365	17,858	14,828
hoextdown/hoedown_fuzzer	8,415	912	9,380	6,171
http-parser/fuzz_parser	95,361	4,693	98,080	64,010
http-parser/fuzz_url	84,443	4,510	93,807	77,950
json-c/tokener_parse_ex_fuzzer	52,064	1,612	55,699	43,174
libpcap/fuzz_both	13,597	3,276	15,930	13,366
libpcap/fuzz_filter	11,169	1,692	14,908	13,852
libpcap/fuzz_pcap	14,048	3,276	14,647	13,959
libtsm/libtsm_fuzzer	6,846	270	6,955	5,532
libyaml/libyaml_fuzzer	17,320	1,491	23,580	23,977
lodepng/lodepng_fuzzer	55,671	2,946	73,405	44,173
lzma/7z_fuzzer	39,016	1,076	40,961	29,775
lzma/filters_fuzzer	10,091	1,008	17,369	16,930
lzma/lzma2dec_fuzzer	7,026	334	12,751	11,988
lzma/lzma2enc_fuzzer	1,771	102	3,872	3,669
lzma/lzmadec_fuzzer	1,804	437	3,602	4,041
lzma/lzmaenc_fuzzer	109	29	305	222
lzma/ppmdenc_fuzzer	1,427	925	2,353	1,783
lzma/xzenc_fuzzer	29	8	82	65
lzo/all_lzo_compress	12,302	1,218	12,846	13123
lzo/lzo_compress_target	65,649	3,524	88,960	43,516
lzo/lzo_decompress_target	103,294	2,471	105,184	60,311
nestegg/fuzz	66,071	3,646	80,161	61,680
nghttp2/nghttp2_fuzzer	40,721	2,270	46,863	24,966
rapidjson/fuzzer	51,166	2,677	60,293	27,393
tinyxml2/xmltest	134,756	2,611	184,365	69,905
yajl-ruby/json_fuzzer	104,143	2,652	155,461	58,587

N-A: Native with ASAN, **W-A**: WASM with ASAN,
N-NA: Native without ASAN, **W-NA**: WASM without ASAN

Table 4: WASM fuzzer performance comparison. Numbers are exec/sec measured with libfuzzer.

`binutils`, `capstone`, and `c-ares`), which are the first four test case in Table 2 with various execution hash collision rates. Our synthesized cheater observes duplicated coverage maps and picks most frequently observed coverage map case and responds with the PoFW answer without actually running the fuzzer. We measured the accumulated rewards of such cheaters under two fuzzing pool configurations: (i) no penalty for the wrong PoFW, (ii) 50% penalty for wrong PoFW. Figure 9 is the result of this evaluation.

The graph shows that 50% penalty reward is sufficient to prevent cheaters goofing against high coverage-map-collision-rate applications (`binutils`, `c-ares` exceed 20% coverage collision rate). Intuitively, higher penalties should render goofing attempts pointless. However, considering that there are edge cases due to non-determinism (Table 3), overly large penalty values should be avoided.

Stashing attack. Stashing is another form of cheating in Fuzzing@Home. It involves a malicious user that honestly participates in PoFW calculation but withholds discovery. Fuzzing@Home's primary mechanism to deal with this issue relies on the principle that a bug or coverage found by a malicious node will quickly be found by another if the participants share input corpus. Theoretically, this is because Fuzzing@Home is built with the explicit design goal of distributing inputs and global coverage information, thus ensuring that there is a high likelihood of another node using the most recent input corpus to find the bug.

In practice, we evaluated how long it takes for one unique bug to be discovered and then consequently re-discovered by other nodes. Table 6 notes the exact dates and times that particular bugs were found during Fuzzing@Home' evaluation. The data indicate that stashing only provides an attacker with prior knowledge of a few minutes to hours of a bug before another node discovers it (e.g., `clamav`). As a result, stashing a discovery will not benefit the attacker because such discoveries will be quickly publicized. Getting validation of the stashed bug from other bug bounty programs requires significantly more time.

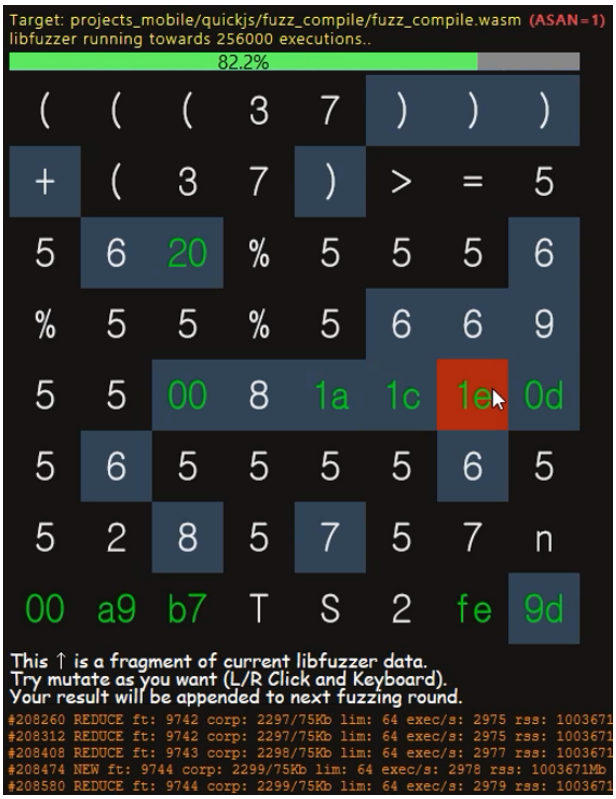

Figure 10: WASM-fuzzer running inside Chrome. The WASM-fuzzer randomly picked one test case and displayed it as a hex-dump. Black tiles are unchanged bytes, and grey tiles are mutated ones by the user.

5.5 WASM-based Fuzzer Performance

WebAssembly (WASM) is useful technology in our project in terms of supporting heterogeneous devices and increasing user accessibility. Figure 10 is a screen capture of Chrome web browser running Fuzzing@Home's fuzzer. Some bugs in our evaluation (`stb`, `quickjs`) were found by WASM-based fuzzers (ASAN enabled).

However, WASM-based fuzzers have the following limitations: (i) limited memory and battery problems in mobile devices, (ii) slower performance compared native execution environments, and (iii) high-cost or infeasible porting. Currently, WASM only supports 32bit virtual memory spaces (due to sandboxing) which is too small for fuzzers, especially with Address Sanitizer (ASAN)'s memory overhead. The memory limitation becomes more challenging in

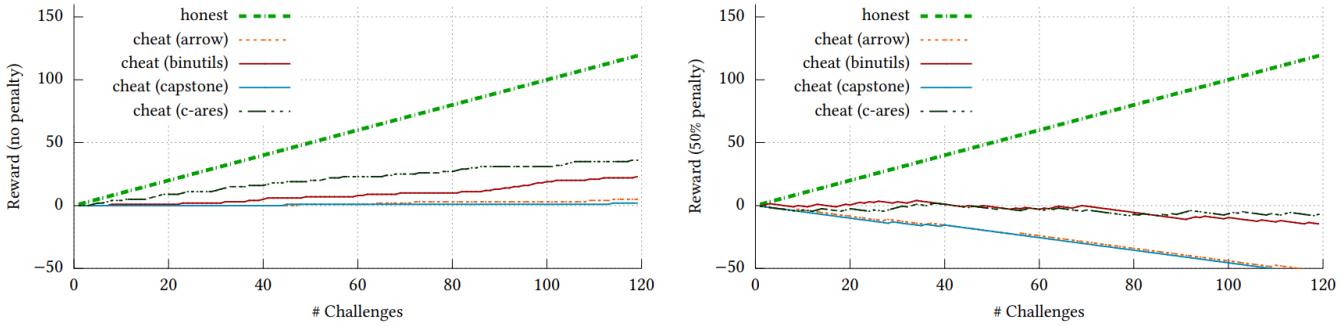

Figure 9: Efficacy of cheating against PoFW challenge. *honest* is the reward amount for honest participants, *cheat* is the reward for attacker in each fuzzing pool; goofing with predicted-hash (highest duplication rate) based on the evaluation result in Table 3. Upon correct answer, the reward is 1 point. The left graph does not impose a penalty for wrong answer while the right side imposes -0.5 point as penalty.

mobile browser environments as their memory limit is more strict (often less than 1GB) compared to desktop browsers; resulting in unpredictable/significant performance slowdown. Due to this limited memory, we only support ASAN for environments that have more memory, such as desktop browsers.

Although WASM ultimately executes a native binary, it is known to incur performance overhead [25]. Table 4 contains a performance comparison of the WASM fuzzer versus Linux-based fuzzers. Average WASM performance without ASAN shows 12% (worst case of 63% in `yajl-ruby/json_fuzzer`) slowdown, and 89% (worst case of 98% in `tinyxml2/xmltest`) slowdown when ASAN is enabled. The result indicates that performance overhead in WASM is primarily due to ASAN which requires significant memory usage. If we consider a massive number of potential participants who join because of the improved accessibility offered by WASM, we believe its performance overhead is somewhat acceptable[3].

Another limitation is the difficulty/impossibility of porting large projects to run with WASM. We attempted to port over 200 projects from OSSFuzz to run on WASM, and succeeded with 30. As large projects often involve complex build systems and multiple dependencies, which can cause build failure or runtime errors, the 30 projects we succeeded in porting to WASM are relatively small ones. With more engineering effort, we estimate it would be possible to port around half of OSSFuzz projects.

6 DISCUSSION AND FUTURE WORK

Sybil Attacks. An attack that distributed networks face is the *Sybil attack*, where a single entity controls a large portion of the network's nodes. Note that the *51% attack* on cryptocurrencies does not apply to Fuzzing@Home because there is no decentralized blockchain to verify. In Fuzzing@Home, globally shared knowledge is entirely controlled by the control server; thus a single node cannot gain rewards without finding bugs or crashes via sheer computing power.

Joining as a software maintainer. To join Fuzzing@Home as software developer who wants to fuzz the application, one has to port their code to run under a Fuzzing@Home-provided fuzzer (libfuzzer or AFL). Additionally, a control server code based on python-flask is needed to manage user authentication and orchestrate fuzzing. At this point, all control servers in our evaluation are managed by us, and all beta-test users are participating as fuzzing clients. In the future, we intend to expand the system to autonomously allow software maintainers to easily join our infrastructure.

User Trust Levels. To reduce the control server overhead, we can differentiate levels of trust for users based on their fuzzing activity. This allows Fuzzing@Home to reduce the amount of PoFW verification, and hence overhead, on nodes that have been known to submit valid PoFW values over time. This would be analogous to banks and credit scores: new account holders are not generally allowed access to higher loans until they make regular payments on smaller ones. Fuzzing@Home could still perform some verification and allow dropping the trust level to account for trustworthy nodes turning malicious.

Towards an Autonomous Ecosystem. Fuzzing@Home ultimately attempts to form an ecosystem, where software maintainers and public users can establish a market. To that end, it requires a system for quantifying the economic value of bugs and automate converting bugs into bug bounty submissions. However, automatically assessing the economic worth of bugs and bug bounty submissions is a hard, if not impossible, problem to solve. Bug bounty incentives are usually holistically reviewed by human reviewers who consider multiple factors [17] like impact, bug type, and quality of the report. Therefore, Fuzzing@Home's reward needs to involve the developers of the software running on the fuzzing pool and their decisions and policies. With sufficient users, we believe software maintainers seeking large computation power for fuzzing can incentivize users by rewarding the fuzzing pool, allowing points to be cashed out.

7 RELATED WORK

Large-scale Fuzzing Infrastructure. Thanks to the excellent parallelizability of fuzzing, many fuzzing infrastructure projects have been created. *Fuzzing-as-a-Service* (FaaS) systems have been launched that allow end-users to fuzz their application without the hassle of physically owning machines or setting up fuzzing software. Google has developed a project called ClusterFuzz [24] that runs on thousands of virtual machines to fuzz their products internally. To improve public security, Google has further released

[3]Currently, protocols in WASM fuzzing pool and native libfuzzer fuzzing pool are incompatible.

OSS-Fuzz [44], donating their computational resources and providing an interface to ClusterFuzz to fuzz open-source software. Moreover, many FaaS systems have been proposed. For example, Microsoft's Project SpringField [33] provides a cloud-based service with its white-box fuzzing. Companies like FuzzBuzz [18] and Fuzzit [19] create platforms for continuous fuzzing: incorporating fuzzing into the development process to discover vulnerabilities proactively. Unlike these projects that use trustworthy machines to perform fuzzing, Fuzzing@Home allows any user to participate in the fuzzing process, even malicious ones.

Crowd-backed Public Computing Projects. Several distributed computing projects have successfully used volunteer computing power in the last two decades. *Folding@Home* targets protein folding. Notably, during the COVID-19 pandemic, hundreds of thousands of users joined the network to push it past 1 exaFLOPS [37]. Another project, *SETI@Home*, ran from 1999 to 2020 and processed astronomical data from telescopes [4] to search for extra-terrestrial life. *distributed.net* cracked DES encryption in 1998 to demonstrate its insecurity [51]. Similar to these projects, Fuzzing@Home can be used as a platform to direct voluntary distributed computing towards public good. However, compared to these volunteer projects, Fuzzing@Home aims to reward contributors with direct/tangible rewards (e.g., the discovery of a crash which could lead to payout or credits on a patch report). Fuzzing@Home hence needs to deal with malicious users who might try to defraud the system for unfair monetary benefits, resulting in various techniques such as *Proof-of-Fuzzing-Work*.

Improving Fuzzing. The majority of previous works in fuzzing focus on better input generation [39] and seed selection. MOPT [32] and AFLFast [7] aim at improving fuzzer performance by improving input selection using power schedules. Concolic and hybrid fuzzers [12] like QSYM [57] and Driller [50] focuses on improving input generation by using constraint solving to get past conditional branches. These fuzzers can satisfy checks that stymie other fuzzers, like checksums or magic numbers. Grammar-based fuzzers try to avoid generating obviously-invalid inputs by constraining inputs to a grammar specification [20, 27, 40, 53] which is useful when fuzzing a language such as Javascript or parsers. CollabFuzz [38] and EnFuzz [11] utilize multiple fuzzers to collaborate for better performance. The collaboration in these works focus on combining multiple different fuzzing algorithms to cover each other shortcomings; while collaboration in our work is focused on expanding the computation scale of a single fuzzing algorithm. Lastly, some fuzzers use machine learning to aid in generating [8, 21, 41, 47] or selecting promising inputs [10, 46]. Instead of focusing on a particular fuzzer's performance, Fuzzing@Home's contribution in this direction focuses on distributed fuzzing, especially utilizing untrusted machines.

8 CONCLUSION

Fuzzing@Home is the first work to support distributed fuzzing based on untrusted heterogeneous clients. Fuzzing@Home allows us to establish a distributed fuzzing network in the form of a public collaborative project. Our system is designed to accommodate untrusted fuzzing participants and also optimize distributed system performance by reducing duplicate computation.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1F1A1049957). This research was also supported, in part, by the NSF award CNS-1563848, CNS-1749711, and ONR under grant N00014-18-1-2662. This work was also supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2022-0-01202, Regional strategic industry convergence security core talent training business).

REFERENCES

[1] 2019. *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA.
[2] 2019. *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
[3] aiohttp. 2019. Async http client/server framework. https://github.com/aio-libs/aiohttp.
[4] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. 2002. SETI@Home: an experiment in public-resource computing. *Commun. ACM* 45, 11 (2002), 56–61.
[5] Apple. 2016. Apple Security Bounty. https://developer.apple.com/security-bounty/.
[6] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 713–724.
[7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. Vienna, Austria.
[8] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. 2018. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 116–122.
[9] Bugcrowd. 2011. Bugcrowd: #1 Crowdsourced Cybersecurity Platform. https://www.bugcrowd.com/.
[10] Yaohui Chen, Mansour Ahmadi, Boyu Wang, Long Lu, et al. 2020. MEUZZ: Smart Seed Scheduling for Hybrid Fuzzing. *arXiv preprint arXiv:2002.08568* (2020).
[11] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Xun Jiao, and Zhuo Su. 2019. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers, See [1].
[12] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Long Lu, et al. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *Proceedings of the 41th IEEE Symposium on Security and Privacy (Oakland)*.
[13] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A Gibson, and Randal E Bryant. 2013. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Farmington, PA.
[14] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, Wil Robertson, Frederick Ulrich, and Ryan Whelan. 2016. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.
[15] Wenliang Du, Jing Jia, Manish Mangal, and Mummoorthy Murugesan. 2004. Uncheatable grid computing. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.* IEEE, 4–11.
[16] Emscripten. 2015. Debugging with Sanitizers. https://emscripten.org/docs/debugging/Sanitizers.html.
[17] Dennis Fisher. 2015. Vupen Founder Launches New Zero-Day Acquisition Firm Zerodium.
[18] FuzzBuzz. 2019. FuzzBuzz: Fuzzing on autopilot. https://fuzzbuzz.io/.
[19] Fuzzit. 2019. Fuzzit - Continuous Fuzzing Made Simple. https://fuzzit.dev/.
[20] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Tucson, AZ.
[21] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Urbana-Champaign, IL.
[22] Philippe Golle and Ilya Mironov. 2001. Uncheatable distributed computations. In *Cryptographers' Track at the RSA Conference*. Springer, 425–440.
[23] Google. 2010. Google Vulnerability Reward Program. https://www.google.com/about/appsecurity/reward-program/.
[24] Google. 2019. ClusterFuzz: a scalable fuzzing infrastructure. https://github.com/google/clusterfuzz.

[25] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.
[26] hackerone. 2012. HackerOne. https://www.hackerone.com/.
[27] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Proceedings of the 21st USENIX Security Symposium (Security)*. Bellevue, WA.
[28] Markus Jakobsson and Ari Juels. 1999. Proofs of work and bread pudding protocols. In *Secure information networks*. Springer, 258–272.
[29] Thomas Kerber, Aggelos Kiayias, Markulf Kohlweiss, and Vassilis Zikas. 2019. Ouroboros crypsinous: Privacy-preserving proof-of-stake, See [2].
[30] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland.
[31] Vivian Lim, Laurie Rubel, Lauren Shookhoff, Mathew Sullivan, and Sarah Williams. 2016. The lottery is a mathematics powerball. *Mathematics Teaching in the Middle School* 21, 9 (2016), 526–532.
[32] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. MOPT: Optimized Mutation Scheduling for Fuzzers, See [1].
[33] Microsoft. 2016. Microsoft Security Risk Detection. https://www.microsoft.com/en-us/security-risk-detection/.
[34] David Molnar, Xue Cong Li, and David A Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs.. In *Proceedings of the 18th USENIX Security Symposium (Security)*. Montreal, Canada.
[35] Pablo Montesinos, Matthew Hicks, Samuel T King, and Josep Torrellas. 2009. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Washington, DC.
[36] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://nakamotoinstitute.org/bitcoin/.
[37] NVIDIA. 2020. Virus War Goes Viral: Folding@home Gets 1.5+ Exaflops to Fight COVID-19. https://blogs.nvidia.com/blog/2020/04/01/foldingathome-exaflop-coronavirus/.
[38] Sebastian Österlund, Elia Geretto, Andrea Jemmett, Emre Güler, Philipp Görz, Thorsten Holz, Cristiano Giuffrida, and Herbert Bos. 2021. Collabfuzz: A framework for collaborative fuzzing. In *Proceedings of the 14th European Workshop on Systems Security*. 1–7.
[39] Sebastian Osterlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. ParmeSan: Sanitizer-guided Greybox Fuzzing. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Boston, MA.
[40] V. Pham, M. Boehme, A. E. Santosa, A. R. Caciulescu, and A. Roychoudhury. 2019. Smart Greybox Fuzzing. *IEEE Transactions on Software Engineering* (2019), 1–1.
[41] Mohit Rajpal, William Blum, and Rishabh Singh. 2017. Not all bytes are equal: Neural byte sieve for fuzzing. *arXiv preprint arXiv:1711.04596* (2017).
[42] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug synthesis: challenging bug-finding tools with deep faults. In *Proceedings of the 26th European Software Engineering Conference (ESEC) and ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. Lake Buena Vista, FL.
[43] Fan Sang, Daehee Jang, Ming-Wei Shih, and Taesoo Kim. 2019. P2FAAS: Toward Privacy-Preserving Fuzzing as a Service. *arXiv preprint arXiv:1909.11164* (2019).
[44] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *Proceedings of the 26th USENIX Security Symposium (Security)*. Vancouver, Canada.
[45] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*. Boston, MA.
[46] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient fuzzing with neural program smoothing, See [2].
[47] Shiqi Shen, Shweta Shinde, Soundarya Ramesh, Abhik Roychoudhury, and Prateek Saxena. 2019. Neuro-Symbolic Execution: Augmenting Symbolic Execution with Neural Constraints.. In *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[48] Yan Shoshitaishvili, Michael Weissbacher, Lukas Dresel, Christopher Salls, Ruoyu Wang, Christopher Kruegel, and Giovanni Vigna. 2017. Rise of the hacrs: Augmenting autonomous cyber reasoning systems with human assistance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 347–362.
[49] Sudarshan M Srinivasan, Srikanth Kandula, Christopher R Andrews, Yuanyuan Zhou, et al. 2004. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC)*. Boston, MA.
[50] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution.. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.
[51] Paul Van De Zande. 2001. The Day DES Died. *SANS Institute, Jul.*
[52] Rijnard van Tonder, John Kotheimer, and Claire Le Goues. 2018. Semantic crash bucketing. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Montpellier, France.
[53] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2017. Skyfire: Data-driven seed generation for fuzzing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, CA.
[54] Maverick Woo, Sang Kil Cha, Samantha Gottlieb, and David Brumley. 2013. Scheduling black-box mutational fuzzing. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*. Berlin, Germany.
[55] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. 2017. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX.
[56] Jiaxi Ye, Bin Zhang, Ruilin Li, Chao Feng, and Chaojing Tang. 2019. Program state sensitive parallel fuzzing for real world software. *IEEE Access* 7 (2019), 42557–42564.
[57] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD.
[58] Zero Day Initiative. 2007. Welcome To Pwn2Own 2020 - The schedule and live results. https://www.thezdi.com/blog/2020/3/17/welcome-to-pwn2own-2020-the-schedule-and-live-results.

A APPENDIX

Table 5 lists the overall numbers of PoFW transactions, coverage expansion reports and bugs discovered in 72 Fuzzing@Home mining pools after 330 days of being deployed. Table 6 is the crash discovery log collected from Fuzzing@Home. Two bugs (null-pointer dereference in `arrow`, out-of-memory in `libtiff`) were discovered from our internal testing environment and not the public deployment. These are excluded from the table. Figure 11 is the entire coverage discovery result of Fuzzing@Home beta service.

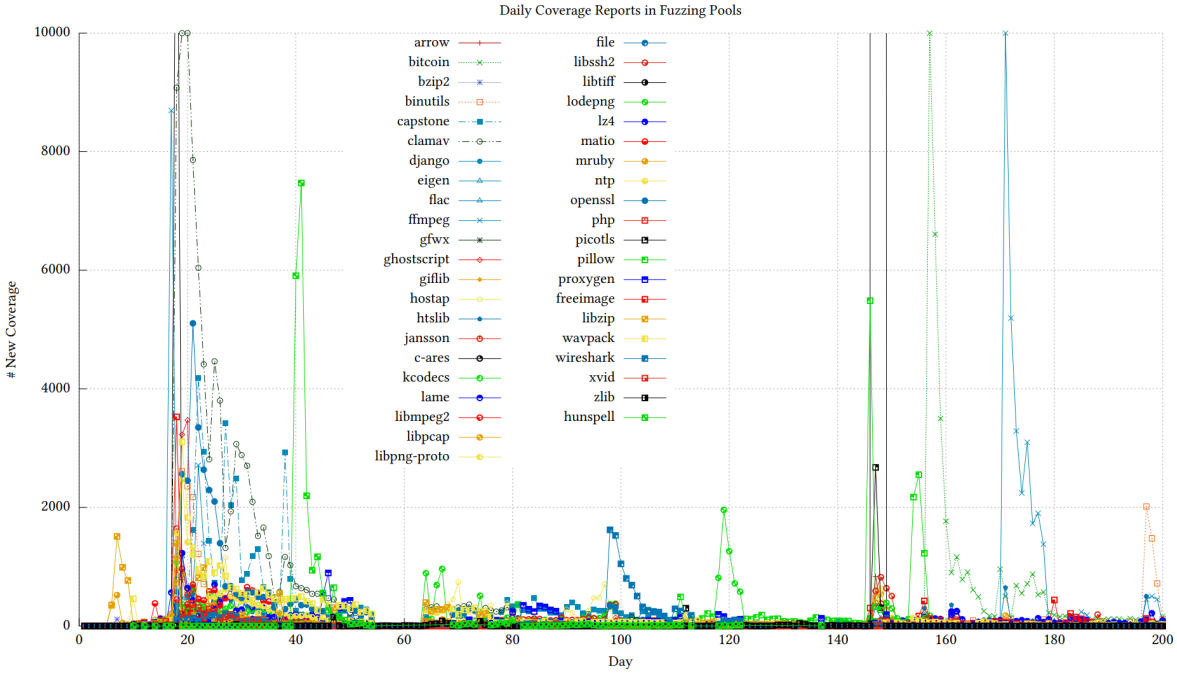

Figure 11: Coverage discovery trend in Fuzzing@Home fuzzing pools since the service's launch date. Data collection was paused around day 54 to perform maintenance on the server infrastructure. In day 143, we performed major version update to fuzzing pools.

Project	libfuzzer LoC	# PoFW	# Coverage	# Crash	Description
arrow	26,722	84,812	5,543	1	Cross-language development platform for Apache in-memory data
avahi	51,611	1,101,050	39,827	0	Apple Zeroconf specification, mDNS, DNS-SD and RFC 3927/IPv4LL
bitcoin	164,735	2,634,954	38,383	0	Open source project which maintains and releases Bitcoin client
bzip2	7,344	13,873	1,256	0	File compression program
binutils	527,834	184,093	13,844	0	GNU collection of binary tools
Capstone	341,292	22,327	44,310	3	Disassembly framework
ClamAV	530,349	1,420,618	97,324	6	Antivirus engine
django	80,394	321,731	1,204	0	High-level Python Web framework
eigen	2,587	156,180	7,310	0	C++ template library for linear algebra
flac	15,793	179,964	80	0	Free Lossless Audio Codec
ffmpeg	8,987,070	203,003	19,902	0	Leading multimedia framework
gfwx	1,607	6,749	4,763	0	Video codec
ghostscript	376,936	193,411	12,930	0	Interpreter for the PostScript language
giflib	1,713	10,484	640	0	GIF image parser library
hostap	254,390	89,541	13,032	0	Implements IEEE 802.11 management, IEEE 802.1X/WPA/WPA2/EAP
htslib	38,339	13,441	10,115	2	C library for high-throughput sequencing data file format
jansson	3,324	20,660	1,661	0	C library for encoding, decoding and manipulating JSON
c-ares	6,164	9,878	783	0	C library for asynchronous DNS requests
kcodecs	8,901	11,314	1,643	0	Collection of methods to manipulate string encodings
lame	20,160	8,995	5,951	0	MPEG Audio Layer III (MP3) encoder
libmpeg2	14,573	13,573	3,575	0	Library for decoding MPEG-1 and MPEG-2
libpcap	82,401	16,976	8,077	0	Library to capture live network data
libpng-proto	79,818	11,533	7,221	0	PNG reference library
libssh2	13,367	136,604	942	0	C library implementing the SSH2 protocol
libtiff	50,958	12,000	11,139	3	Library for Tag Image File Format (TIFF)
lodepng	5,264	15,756	1,605	0	PNG image decoder and encoder
lz4	32,769	9,682	16,166	0	Lossless compression algorithm
matio	11,162	596,220	14,766	118	C library for reading and writing binary MATLAB MAT files
mruby	34,533	13,805	15,328	3	Lightweight implementation of the Ruby language
ntp	59,152	12,992	523	0	Protocol designed to synchronize the clocks over a network
openssl	506,730	137,110	18,911	0	Implementation of TLS and SSL protocols
php	1,508,724	165,799	40,142	0	General-purpose scripting language
picotls	28,769	15,772	889	0	TLS 1.3 (RFC 8446) implementation written in C
pillow	38,442	21,973	1,126	0	Python Imaging Library
proxygen	27,652	12,031	5,315	0	C++ HTTP abstractions used at Facebook
freeimage	223,485	561,804	27,861	46	Library for image formats including PNG, BMP, TIFF and others
libzip	6,744	10,803	3,036	0	C library for reading, creating, and modifying zip archives
wavpack	5,007	88,338	9,703	0	Audio compression format
wireshark	4,353,042	168,741	36,094	0	Network packet analyzer
xvid	32,250	99,080	13,245	2	Video codec strong in compression
zlib	26,320	10,004	3,853	0	Library used for data compression
hunspell	11,650	3,855	143	0	Spell checker of LibreOffice, Mozilla Firefox 3 and others
samba	739,401	2,095,505	20,398	6	Windows interoperability suite of programs for Linux
brotli (wasm)	20,614	247	1,378	0	Generic-purpose lossless compression algorithm
bzip2 (wasm)	7,344	1,431	3,780	0	File compression program
gfwx (wasm)	1,607	32,708	898	1281	Video codec
guetzli (wasm)	10,561	344	1,708	0	JPEG encoder that aims for excellent compression
haproxy (wasm)	174,463	3,489	2,784	0	Reliable, High Performance TCP/HTTP Load Balancer
hoextdown (wasm)	9,574	740	3,134	0	A fully functional (X)HTML renderer
http-parser (wasm)	8,125	22,554	30,653	0	This is a parser for HTTP messages written in C
json-c (wasm)	10,404	1,699	13,940	0	JSON-C implements a reference counting object model
libexif (wasm)	13,828	2,674	950	53	libexif is a library for parsing, editing, saving EXIF data
libfdk-aac (wasm)	165,957	103,738	6,906	0	library for encoding and decoding Advanced Audio Coding
libpcap (wasm)	82,401	5,646	2,838	0	Library to capture live network data
libtsm (wasm)	10,331	7,693	5,263	391	Terminal-emulator State Machine
libucl (wasm)	15,980	81	443	0	Libucl: Universal configuration library parser
libyaml (wasm)	14,309	2,061	5,181	0	C library for parsing and emitting YAML
lodepng (wasm)	15,237	102	676	0	PNG image decoder and encoder
lzma (wasm)	89,340	12,601	25,560	0	Lempel-Ziv-Markov chain lossless compression algorithm
lzo (wasm)	18,276	7,808	7,261	547	Lossless data compression library written in ANSI C
nestegg (wasm)	4,251	2,276	3,608	0	Open source commerce solution for Ruby on Rails
nghttp2 (wasm)	49,753	1,077	1,531	0	Implementation of HTTP/2 and its header compression
piex (wasm)	3,138	268	114	0	Preview Image Extractor
quickjs (wasm)	77,969	4,474	4,717	0	Small and embeddable Javascript engine
rapidjson (wasm)	17,405	1,197	8,461	0	JSON parser and generator for C++
skcms (wasm)	5,962	4,553	3,705	0	Open source 2D graphics library
sqlite3 (wasm)	179,344	145	389	0	C-language library that implements SQL database engine
stb (wasm)	56,774	904	1,613	85	Single-file public domain image libraries for C/C++
tinyxml2 (wasm)	5,624	601	501	0	Simple, small, efficient, C++ XML parser
uriparser (wasm)	11,341	2,676	9,336	0	RFC 3986 compliant URI parsing library
yajl-ruby (wasm)	3,610	705	3,762	0	YAJL C Bindings for Ruby
zopfli (wasm)	5,583	38	67	0	Library to perform very good, but slow, zlib compression

Table 5: Summary of the projects targeted and data collected during Fuzzing@Home's real-world deployment. We can deduce the amount of user activity based on # PoFW (typically single PoFW requires a few minutes of fuzzing). Note that # coverage can be greater than # PoFW because a single PoFW can report multiple coverages. Crash numbers are before manual analysis based de-duplication is applied. Crashes in WASM pools (`libexif, libtsm, lzo`) except `stb`, are caused by address sanitizer false positive.

No	Discovery Time (Day - HH:MM:SS)	Project	Description (based on addresss sanitizer)
1	Day 04 - 20:58:38	matio	allocation-size-too-big
2	Day 04 - 20:59:51	matio	out-of-memory
3	Day 04 - 23:23:24	matio	out-of-memory
4	Day 12 - 19:06:38	matio	SEGV on unknown address at code location 0x8dd
5	Day 12 - 19:06:42	matio	allocation-size-too-big
6	Day 12 - 19:07:57	matio	allocation-size-too-big
7	Day 12 - 21:51:23	matio	out-of-memory
8	Day 12 - 21:51:27	matio	out-of-memory
9	Day 12 - 21:51:38	matio	allocation-size-too-big
10	Day 12 - 21:51:39	matio	SEGV on unknown address at code location 0xcde
11	Day 12 - 21:51:40	matio	out-of-memory
12	Day 12 - 21:52:00	matio	heap-buffer-overflow (read of size 1) at code location 0x313
13	Day 12 - 21:53:00	matio	allocation-size-too-big
14	Day 12 - 21:53:14	matio	out-of-memory
15	Day 12 - 21:53:47	matio	out-of-memory
16	Day 12 - 21:55:20	matio	out-of-memory
17	Day 12 - 21:55:34	matio	out-of-memory
18	Day 12 - 21:56:29	matio	calloc-overflow at code location 0x0b2
19	Day 12 - 21:56:37	matio	out-of-memory
20	Day 12 - 21:57:00	matio	SEGV on unknown address at code location 0x045
21	Day 12 - 21:57:30	matio	allocation-size-too-big
22	Day 12 - 21:58:33	matio	out-of-memory
23	Day 12 - 22:00:09	matio	allocation-size-too-big
24	Day 12 - 22:04:32	matio	out-of-memory
25	Day 12 - 22:18:53	matio	SEGV on unknown address at code location 0x045
26	Day 12 - 22:20:08	matio	allocation-size-too-big
27	Day 12 - 23:02:30	matio	out-of-memory
28	Day 12 - 23:24:35	matio	heap-buffer-overflow (read of size 45) at code location 0x766
29	Day 12 - 23:24:41	matio	allocation-size-too-big
30	Day 12 - 23:25:06	matio	stack-buffer-overflow (write of size 8) at code location 0x8a8
31	Day 12 - 23:25:48	matio	allocation-size-too-big
32	Day 12 - 23:26:02	matio	heap-buffer-overflow (read of size 65334) at code location 0x766
33	Day 12 - 23:26:51	matio	SEGV on unknown address at code location 0xb15
34	Day 12 - 23:34:44	matio	allocation-size-too-big
35	Day 12 - 23:38:23	matio	allocation-size-too-big
36	Day 12 - 23:40:26	matio	allocation-size-too-big
37	Day 13 - 00:00:42	matio	allocation-size-too-big
38	Day 13 - 00:05:15	matio	calloc-overflow at code location 0x0b2
39	Day 13 - 00:05:52	matio	allocation-size-too-big
40	Day 13 - 00:11:54	matio	allocation-size-too-big
41	Day 13 - 00:38:07	matio	heap-buffer-overflow (read of size 1) at code location 0x313
42	Day 13 - 01:16:44	matio	allocation-size-too-big
43	Day 13 - 01:22:15	matio	out-of-memory
44	Day 13 - 02:54:42	matio	out-of-memory
45	Day 13 - 02:56:55	matio	allocation-size-too-big
46	Day 13 - 03:01:08	matio	allocation-size-too-big
47	Day 13 - 06:35:26	matio	allocation-size-too-big
48	Day 13 - 07:25:46	matio	calloc-overflow at code location 0x0b2
49	Day 13 - 10:25:13	matio	calloc-overflow at code location 0x0b2
50	Day 13 - 11:40:48	clamav	heap-buffer-overflow (read of size 1) at code location 0x9ef
51	Day 13 - 12:15:07	clamav	heap-buffer-overflow (read of size 1) at code location 0x9ef
52	Day 13 - 12:31:10	clamav	heap-buffer-overflow (read of size 1) at code location 0x9ef
53	Day 13 - 15:01:59	matio	allocation-size-too-big
54	Day 13 - 21:06:27	matio	heap-buffer-overflow (read of size 321) at code location 0x638
55	Day 14 - 01:00:09	matio	heap-buffer-overflow (read of size 2130706432) 766
56	Day 14 - 03:09:20	matio	SEGV on unknown address at code location 0xcde
57	Day 14 - 03:36:53	matio	SEGV on unknown address at code location 0x045
58	Day 14 - 04:07:34	matio	SEGV on unknown address at code location 0x045
59	Day 14 - 04:09:40	matio	allocation-size-too-big
60	Day 14 - 06:05:41	matio	SEGV on unknown address at code location 0x01a
61	Day 14 - 06:06:30	matio	allocation-size-too-big
62	Day 14 - 06:58:44	matio	SEGV on unknown address at code location 0x045
63	Day 14 - 07:18:17	matio	allocation-size-too-big
64	Day 14 - 08:03:26	clamav	heap-buffer-overflow (read of size 1) at code location 0x9ef
65	Day 14 - 09:05:06	matio	allocation-size-too-big
66	Day 14 - 10:10:16	matio	SEGV on unknown address at code location 0x045
67	Day 14 - 10:32:58	matio	heap-buffer-overflow (read of size 6) 766
68	Day 14 - 10:34:36	matio	out-of-memory
69	Day 14 - 11:27:58	clamav	heap-buffer-overflow (read of size 1) at code location 0x9ef
70	Day 14 - 14:29:44	matio	out-of-memory
71	Day 14 - 14:48:29	matio	SEGV on unknown address at code location 0x01a
72	Day 14 - 15:43:07	matio	SEGV on unknown address at code location 0x0be
73	Day 14 - 18:08:03	matio	allocation-size-too-big
74	Day 14 - 20:57:41	matio	heap-buffer-overflow (write of size 48) at code location 0x8a8
75	Day 14 - 21:53:07	matio	heap-buffer-overflow (read of size 1) at code location 0x638
76	Day 14 - 22:10:21	matio	allocation-size-too-big
77	Day 15 - 02:40:19	matio	memcpy-param-overlap
78	Day 15 - 07:54:55	matio	allocation-size-too-big
79	Day 15 - 09:54:52	matio	allocation-size-too-big
80	Day 15 - 10:23:37	matio	heap-buffer-overflow (read of size 1) at code location 0x313
81	Day 15 - 10:47:01	matio	SEGV on unknown address at code location 0x01a
82	Day 15 - 12:58:56	matio	out-of-memory
83	Day 16 - 07:57:21	capstone	global-buffer-overflow
84	Day 16 - 08:13:47	capstone	global-buffer-overflow
85	Day 16 - 12:21:00	matio	heap-buffer-overflow (read of size 1) at code location 0x313
86	Day 16 - 21:57:08	matio	out-of-memory
87	Day 16 - 22:35:57	matio	SEGV on unknown address in libc
88	Day 19 - 07:21:21	matio	null pointer dereference in libc
89	Day 20 - 04:18:29	matio	allocation-size-too-big
90	Day 20 - 10:03:45	matio	allocation-size-too-big
91	Day 20 - 16:19:04	matio	SEGV on unknown address in libc
92	Day 20 - 20:48:45	matio	out-of-memory
93	Day 21 - 08:49:06	matio	heap-buffer-overflow (read of size 1) at code location 0x313
94	Day 21 - 10:16:41	matio	floating point exception
95	Day 21 - 16:37:10	matio	SEGV on unknown address at code location 0x450
96	Day 21 - 17:16:11	matio	heap-buffer-overflow (read of size 49) at code location 0x638
97	Day 21 - 22:20:05	matio	null pointer dereference in libc
98	Day 22 - 04:47:23	matio	SEGV on unknown address at code location 0x1a8
99	Day 22 - 08:41:33	matio	out-of-memory
100	Day 22 - 14:45:44	matio	heap-buffer-overflow (read of size 1) at code location 0x313
101	Day 23 - 14:30:19	htslib	out-of-memory
102	Day 24 - 00:21:36	matio	null pointer dereference at code location 0xf3d
103	Day 24 - 07:42:36	matio	out-of-memory
104	Day 25 - 09:02:39	matio	null pointer dereference in libc
105	Day 26 - 01:20:51	matio	null pointer dereference in libc
106	Day 26 - 05:35:44	matio	allocation-size-too-big
107	Day 26 - 11:11:26	matio	null pointer dereference at code location 0xf3d
108	Day 26 - 23:22:13	matio	null pointer dereference at code location 0x59f
109	Day 28 - 08:35:20	matio	null pointer dereference in libc
110	Day 28 - 15:00:16	matio	allocation-size-too-big
111	Day 28 - 18:44:53	freeimage	stack-buffer-overflow (read of size 129) at code location 0x0d3
112	Day 28 - 20:31:37	clamav	null pointer dereference in libc
113	Day 28 - 23:24:17	matio	allocation-size-too-big
114	Day 29 - 16:32:40	matio	allocation-size-too-big
115	Day 30 - 04:45:07	matio	null pointer dereference in libc libc
116	Day 33 - 01:23:55	matio	allocation-size-too-big
117	Day 36 - 03:45:53	samba	heap-buffer-overflow (read of size 1) at code location 0xe27
118	Day 40 - 03:11:04	freeimage	out-of-memory
119	Day 41 - 04:02:43	freeimage	allocation-size-too-big
120	Day 41 - 09:22:29	freeimage	allocation-size-too-big
121	Day 41 - 16:50:06	freeimage	out-of-memory
122	Day 41 - 18:56:14	freeimage	allocation-size-too-big
123	Day 41 - 19:37:24	freeimage	allocation-size-too-big
124	Day 41 - 20:16:12	freeimage	out-of-memory
125	Day 42 - 06:15:58	freeimage	allocation-size-too-big
126	Day 42 - 07:16:33	freeimage	allocation-size-too-big
127	Day 42 - 07:35:40	freeimage	heap-buffer-overflow (write of size 17) at code location 0x33f
128	Day 42 - 21:10:28	freeimage	allocation-size-too-big
129	Day 43 - 08:25:40	freeimage	allocation-size-too-big
120	Day 46 - 21:27:20	freeimage	allocation-size-too-big
131	Day 66 - 07:17:55	matio	out-of-memory
132	Day 67 - 03:18:40	matio	null pointer dereference in libc
133	Day 72 - 13:49:41	freeimage	allocation-size-too-big
134	Day 74 - 12:44:22	freeimage	out-of-memory
135	Day 76 - 19:27:22	freeimage	allocation-size-too-big
136	Day 76 - 22:48:25	freeimage	allocation-size-too-big
137	Day 78 - 16:51:46	freeimage	allocation-size-too-big
138	Day 78 - 00:55:15	freeimage	allocation-size-too-big
139	Day 79 - 15:10:13	mruby	allocation-size-too-big
140	Day 83 - 05:27:07	freeimage	allocation-size-too-big
141	Day 89 - 12:16:04	freeimage	allocation-size-too-big
142	Day 89 - 13:25:42	mruby	allocation-size-too-big
143	Day 99 - 07:22:49	xvid	heap-buffer-overflow (read) at code location 0xe27
144	Day 104 - 12:15:30	freeimage	allocation-size-too-big
145	Day 104 - 19:21:47	freeimage	allocation-size-too-big
146	Day 105 - 10:10:12	xvid	heap-buffer-overflow (read) at code location 0x5df
147	Day 109 - 12:38:21	freeimage	allocation-size-too-big
148	Day 110 - 00:41:47	freeimage	allocation-size-too-big
149	Day 110 - 08:22:12	freeimage	allocation-size-too-big
150	Day 116 - 00:50:46	freeimage	allocation-size-too-big
151	Day 116 - 01:58:21	freeimage	allocation-size-too-big
152	Day 117 - 01:07:36	freeimage	allocation-size-too-big
153	Day 117 - 18:14:26	freeimage	stack-buffer-overflow (read of size 243) at code location 0x0e2
154	Day 118 - 22:43:54	freeimage	heap-buffer-overflow (write of size 4) at code location 0x9af
155	Day 119 - 07:59:29	freeimage	allocation-size-too-big
156	Day 124 - 17:21:46	freeimage	allocation-size-too-big
157	Day 133 - 07:04:46	freeimage	out-of-memory
158	Day 140 - 20:16:49	freeimage	allocation-size-too-big
159	Day 140 - 20:27:14	freeimage	allocation-size-too-big
160	Day 140 - 20:46:13	freeimage	allocation-size-too-big
161	Day 140 - 21:40:35	freeimage	allocation-size-too-big
162	Day 140 - 21:45:52	freeimage	allocation-size-too-big
163	Day 141 - 11:22:39	samba	heap-buffer-overflow (read of size 1) at code location 0x5df
164	Day 142 - 15:18:34	clamav	null pointer dereference in libc
165	Day 143 - 13:23:23	freeimage	global-buffer-overflow (read of size 9) at code location 0x5ce
166	Day 148 - 05:24:43	stb	heap-buffer-overflow (read of size 3) at code location 0x051
167	Day 149 - 06:19:33	htslib	out-of-memory
168	Day 152 - 17:47:25	capstone	global-buffer-overflow

Table 6: Crash discovery log from Fuzzing@Home's public deployment. The code location is the lower 12 bits of the faulting program counter.