Towards Automated Vulnerability Scanning of Network Servers

Nathan Schagen
Vrije Universiteit Amsterdam
nschagen@gmail.com

Koen Koning
Vrije Universiteit Amsterdam
koen.koning@vu.nl

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

ABSTRACT

We explore a new technique for safe patch fingerprinting to automate vulnerability scanning of network servers. Our technique helps automate the discovery of inputs that safely discriminate vulnerable from patched servers for the latest vulnerabilities. This enables rapid updates to vulnerability scanning tools as new software vulnerabilities are discovered, allowing administrators to scan and secure their networks more quickly. To ensure such scans are safe and ethical, we need to reject inputs with malicious side effects.

We have implemented a framework, based on delta execution, which tests the discriminative property of such inputs, as well as their safety. We use a fuzzer to find promising candidate inputs to further automate the process. To illustrate the potential of this approach, we present a Heartbleed case study.

CCS CONCEPTS

• **Security and privacy → Network security**;

KEYWORDS

Vulnerability fingerprinting, Internet-wide scanning, Network security

ACM Reference Format:
Nathan Schagen, Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2018. Towards Automated Vulnerability Scanning of Network Servers. In *EuroSec'18: 11th European Workshop on Systems Security , April 23–26, 2018, Porto, Portugal.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3193111.3193116

1 INTRODUCTION

Vulnerability scanners allow administrators and researchers to prove that a host is vulnerable to certain attacks. Most of such scanners focus on web-applications, since many web-exploits (e.g, SQL injection and cross-site-scripting) can be detected using a small set of inputs. These vulnerabilities arise due to lack of input sanitization and the scanner relies on well-known web technologies to ensure the vulnerability reveals itself.

Compiled server applications, on the other hand, are more difficult to scan, as these are not built using a common, interpreted technology stack. Scanners must be equipped with vulnerability-specific scripts, which take more time to develop and test. For example, ZMap [3] offers a script specifically to scan for Heartbleed, which is not trivial to develop.

In this paper, we investigate a novel method we call *safe patch fingerprinting*, which aims to fingerprint behavioral changes of a server as a result of applying a security patch. More specifically, we want to discover specific inputs which provoke a response that allows us to remotely discriminate between patched and unpatched versions. We shall call such inputs *discriminators*.

Automating the fingerprinting process will allow vulnerability scanners to be updated quickly, allowing organizations to assess vulnerability impact and take necessary precautions. Alternatively, automated fingerprinting can be used when developing a security patch to ensure it is *seamless* and thus cannot be fingerprinted.

Because this method is patch-based, its detection capabilities generalize across many versions of the vulnerable server. This is important, because when patches are backported, many major versions end up having both vulnerable and patched builds, making version information an unreliable indicator of vulnerabilities.

We must ensure that such scans do not harm production servers, especially when using the technique on a large scale. This is challenging, given that many different versions of a server can respond differently to particular stimuli. Hence, automation is essential as it allows extensive testing before use in production.

In this research, our main focus is on *discriminator testing*, which involves testing whether a given input to a server allows us to discriminate vulnerable from patched builds in a safe manner. Besides this, we have taken an initial step towards automated *discriminator discovery* using a fuzzer.

This work represents a first step. Once the technique has matured, it could be used for vulnerability scanning, even at the Internet scale. ZMap [3] offers an infrastructure to quickly scan a large number of hosts, which—combined with our technique—could allow us to measure patch adoption over time.

In Section 2 we given an overview of the system and related terminology. In Section 3 we discuss the design of the *discriminator testing* and *discriminator discovery* systems. The implementation of the former is discussed in Section 4, followed by the evaluation in section 5. Some related work can be found in Section 6. Finally, we discuss some findings and conclude in Sections 7 and 8 respectively.

EuroSec'18, April 23–26, 2018, Porto, Portugal

ACM ISBN 978-1-4503-5652-7/18/04.
https://doi.org/10.1145/3193111.3193116

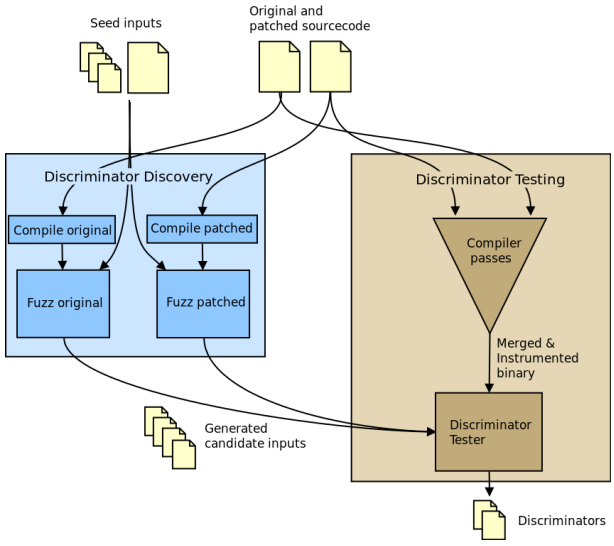

Figure 1: Overview of the patch fingerprinting framework.

2 OVERVIEW

This paper we primarily focus on *discriminator testing*, a method to decide whether an input safely discriminates the patch and unpatched version. Finding such discriminators requires complex constraints to be satisfied, as discussed later. We also extend our system for automated discovery of discriminators using the test method we have devised. The whole system is shown in Figure 1.

Given a vulnerability and its patch, we compile both the original and the patched version of the server. A fuzzer is then used on each of the compiled binaries, to generate a set of inputs that may be discriminators. We bootstrap the fuzzing process using a number of *seed inputs*. These could either be known or could be generated using an exploit script. Once candidate inputs are generated, we can test whether any of them is a discriminator. We do this by passing them into an instrumented server, which uses a technique called *delta execution* [10] to analyze how both versions of the server behave. The inputs that are valid discriminators are returned at the end.

Patches usually affect groups of source code lines, which are called *hunks*. These lines correspond to groups of compiled instructions which we refer to as *patch-sites*.

Discriminators are inputs having the following properties:

- Program execution must **reach** a *patch-site*. In other words, the input must cause added or modified instructions to be executed.
- The input must **infect** the program state. This means that there must be memory state divergence between the execution of the original and patched build of a server. Memory bytes that differ between these executions are said to be *infected*. This infection can spread as data is copied, combined or used in branching conditions. Infections can be removed when the bytes converge to the same value.
- The *infected program state* must **propagate** to the output, such that the difference between the patched and unpatched server externally visible and thus observable by the vulnerability scanner. Inputs that have this property are said to be *propagating*.
- It must be **safe**, meaning that input does not exploit the vulnerability in a harmful way. What this means depends on both the vulnerability and the server. Therefore, the safety requirements must be established by an analyst.

To summarize, we need to test whether an input exercises the vulnerable code, resulting in a state divergence causing an observable effect, discriminating the two versions, without actually exploiting the vulnerability.

2.1 Safety

For a discriminator to be usable in real-world scenarios, it must be safe to use. As stated before, what safety is depends on the context. We define two broad types of safety to be used for two types of vulnerabilities: *Integrity* means that the program state was not corrupted. In other words, no modified state must cause the program to malfunction or behave on behalf of the attacker. *Confidentiality* means that no internal information is leaked to the attacker. This distinction is important as both require different methods of ensuring safety.

Besides the type of vulnerability, safety also depends on characteristics of the server. For example, crashing a process is normally considered unsafe. However, some servers, such as Nginx and Apache, will automatically fork off new worker processes once this happens, undoing the damage that was done. One can argue that process crashes are safe in this scenario. Alternatively, a buffer-overflow vulnerability might only be detectable remotely if we attempt an overflow, forcing us to overflow at least a single byte. Depending on the memory layout, this could be harmless or very damaging. Because modern compilers add padding to data structures, small overflows may be perfectly safe.

2.2 Propagation

We want to *leak* one bit of information: whether the patch is applied to the server or not. Besides regular I/O, other channels may exist through which we can leak this bit, such as timing side-channels. These are outside the scope of this work.

As opposed to *reachability* and *infection*, it is not known whether propagating inputs exist for a vulnerability. When the vulnerable code has no information flows that lead back to the network, no discriminator exists and we will not be able to scan for it. In our work, we focus on determining whether a given input propagates, rather than efficiently finding propagating inputs, for arbitrary vulnerabilities.

In this work, we only consider propagation of infected state towards the `write`-family of system calls. Other I/O functions can easily be supported. Propagation can happen because both the original and patched version send a different data buffer, which we call *propagation-through-difference* or because only one version sends a response at all (*propagation-through-absence*). In the latter case, we should repeat the request a few times to ascertain ourselves that the lack of a response is not caused by host or network failure.

Original echo server

```
1  typedef struct {int size; char buf[99];} echo_t;
2
3  void handle_request(int clientfd) {
4    echo_t *e = malloc(sizeof(echo_t));
5    read(clientfd, &e->size, sizeof(int));
6    read(clientfd, e->buf, e->size);
7    write(clientfd, e->buf, e->size);
8    free(e);
9  }
10
11
```

Patched echo server

```
typedef struct {int size; char buf[99];} echo_t;

void handle_request(int clientfd) {
  echo_t *e = malloc(sizeof(echo_t));
  read(clientfd, &e->size, sizeof(int));
  if (e->size > 0 && e->size <= 99) {
    read(clientfd, echo->buf, e->size);
    write(clientfd, echo->buf, e->size);
  }
  free(echo);
}
```

Figure 2: Echo server contains a buffer overflow vulnerability. It can be safely detected using a small overflow that only overwrites padding bytes

2.3 Example

In Figure 2, we demonstrate an echo server, which has a simple buffer overflow vulnerability. By providing a *size* that is larger than 99, we can overflow the buffer on the heap.

To discriminate, we are forced to overflow the buffer and see how the server responds. Fortunately, compilers align structures in memory to at least 4-byte boundaries. Thus, *echo_t* has a size of 104 bytes. Therefore, it is safe to send 100 bytes, without corrupting the server. This is a good example of *propagation-through-absence* as the patched server will not respond when a 100 bytes are sent to it.

3 DESIGN

3.1 Discriminator testing

Finding out whether a given input is a discriminator could be done by running both the patched and the unpatched builds and see if a particular input leads to different observable outcomes. However, each execution of the server may be different due to non-determinism, timing and other factors, which complicates discovery testing. Furthermore, it is impossible to rule out memory corruption, as this may only become evident under specific circumstances. We could compare the state of both executions to inspect diverged state. As we assume the patched version is not vulnerable, having identical program state implies that the unpatched version is not corrupted.

Unfortunately, comparing the state is difficult as there are many reasons for divergence, besides execution of a patch-site. Timing, randomization, thread-interleaving, interactions with the OS, and subtle differences in binary layout caused by the patch can all result in spurious state differences. To properly isolate the effects of the patch, all this noise and non-determinism must be eradicated.

We have implemented *delta execution* [10], which allows us to only run a *patch-site* of the program (*delta code*) in parallel. The common code is running as a single execution (*merged execution*). When we reach delta code, we split off into two different executions. We are now running a *split execution*. These executions will continue to run, and state differences introduced by delta code can be easily extracted. At specific moments, we attempt to *merge*, which means that we compare the state of the two executions. When there are no differences, we drop one of the executions and let the other continue as a *merged execution*.

Delta execution has a number of advantages. First of all, it runs a single execution most of the time, eradicating most diverged state that would otherwise appear between separate runs of the server. The diverged state we observe will thus be a result of the patch. This will only work when a small part of the program was modified by the patch. Fortunately, security patches tend to be small [10] so we can stay in merged execution most of the time. Secondly, when merging is successful, all infected state is gone, giving strong safety guarantees. Finally, delta execution helps us find propagating inputs, because we can easily interpose I/O calls during split execution and observe any differences.

Besides integrity, we must also address the problem of confidentiality in case of information disclosure vulnerabilities. This is an extremely difficult problem to fully solve, as it would require data flow analysis and annotating data structures as sensitive. Therefore, we assume that information disclosure vulnerabilities are exploited using controlled memory copy (such as *memcpy()*), which will violate data structure boundaries and may read uninitialized data. This allows us to use the AddressSanitizer (ASan) [8] which detects out-of-bounds access to stack, heap and globals, and the MemorySanitizer (MSan) [9] to catch *uninitialized reads* (UMR). Our delta execution framework can run with either one of these sanitizers. It is recommended to run both when working with an information disclosure vulnerability.

3.2 Discriminator discovery

To make *discriminator discovery* feasible, we assume that one or more *seed inputs* are known that exercise the patched code. In other words, they must have the *reachability* and *infection*. These could be generated by an available exploit script but could also be crashing inputs found by fuzzing. They will not be *safe* as they result in a crash or even full exploitation. Also it is unknown whether they will *propagate*.

We use VUzzer [7], which is a mutation-based evolutionary fuzzer to mutate these seed inputs. The intuition is that discriminators look a lot like these malicious inputs, except that they will not result in a crash or exploitation.

VUzzer was designed to work with locally executed programs that accept input via `STDIN`. A single `read()` would consume input, which VUzzer would taint and track during execution. Bytes, at specific offsets, that are used by `CMP` instructions are targeted by the fuzzer to maximize basic-block coverage, which is trivial when the other `CMP` operand is a constant.

We have made a few modifications to the fuzzer to make it work for servers. Instead of sending input to `STDIN`, an application-specific setup script will start the server, establish a connection and carry out initial interactions such as authentication. Finally, the fuzzer-generated payload will be sent to the server and VUzzer will start taint-tracking on the first `read()` call that is executed. We also made VUzzer maximize basic-block coverage inside the patched function rather than the entire application.

4 IMPLEMENTATION

Our implementation uses the LLVM infrastructure [5] to automatically transform existing C/C++ applications. In this section, we will discuss the two most important components in our delta execution framework, which are the *deltafy compiler pass*, which merges two almost identical LLVM bitcode files, and the *libdelta* shared library, which coordinates delta execution during run-time at process granularity.

4.1 Deltafy pass

The deltafy pass takes two LLVM bitcode files, which represent our original and patched version. First it compares all functions to find the one(s) modified by the patch. It then moves the patched versions of all modified functions into the original bitcode file and introduces a *proxy function* for each affected function. This proxy function acts as a switch between the different versions of the function, passing on all arguments and returning the return value. When the proxy function is called, it will call the *split hook*, whose return value determines whether the original or patched version of the function will be called next. This *split hook* is implemented in *libdelta* and calls `fork()`. The parent and child process each return a different value from the hook, causing both the original and patched version to execute in parallel.

Note that we split as soon as the function containing the *patch-site* is called. This does not mean that we execute the patch site. Therefore, we may be in split execution longer than strictly necessary.

Delta execution only supports modifications to the code. No global variables can be introduced and no structs may be changed. This is not problematic since most security patches only involve small code changes [10].

4.2 Libdelta

Libdelta coordinates delta-execution. It does bookkeeping for each process, allowing each to split and merge individually as patch-sites are executed. It carries out splitting when the function containing the *patch-site* is called, which is implemented using `fork()` as described above.

When running in split state, a merge is attempted when a function returns. The intuition is that some task is finished, state cleared and that we can thus test for state convergence. Hooks are installed on function returns using a separate compile pass. *Libdelta* will wait for both the original and patched process to arrive at a function return, before carrying out a full comparison of the memory state.

Comparing the memory state of two processes is an expensive operation. We have optimized it by leveraging *soft-dirty-bits* [1] to see which pages of memory were modified. Only these modified pages are compared byte-by-byte. Differences are written to a file. When there are none, the merge is successful and the patched execution (child process) `exit()`'s and libdelta goes back to merged execution. When a merge fails, it is attempted a number of times on subsequent function returns. If the number of failed merges exceeds a threshold, libdelta terminates all running processes and concludes the input is not safe.

Libdelta needs to do I/O virtualization when in split execution, because we do not want each I/O function to be executed twice; the outside world should only perceive the effects of a single process. On the other hand, the effects on process memory must be replicated to both processes. *Libdelta* thus interposes a wide range of `libc` functions to manage all interactions. Secondly, I/O calls are intercepted to detect propagation of infected state. Currently, we only compare buffers sent to system calls of the `write`-family and detect calls that are not matched by the other process, indicating *propagation-through-absence*

Finally, *libdelta* supports whitelisting of both global symbols and allocated or `mmap`'ed regions. For the latter, the server that is under analysis must be modified to call the whitelist function exposed by libdelta.

5 RESULTS

At this point, we have only evaluated our framework with the Heartbleed vulnerability. We were unable to test against other vulnerabilities because often a working patch or suitable seed inputs were not available. We have also crafted a number of test servers, which we use to demonstrate the technique. We discuss the outcomes of these experiments as preliminary results.

5.1 Heartbleed

Heartbleed (CVE-2014-0160) is a critical information disclosure vulnerability, which enables hackers to leak memory of an OpenSSL server. It was one of the main drivers behind this research because it contains a known discriminator, making it a good ground-truth to benchmark our framework. Our testing framework correctly identifies this ground truth as valid discriminator.

Furthermore, we assessed the effectiveness of the sanitizers to detect information leakage. The *address sanitizer* (ASan) allowed us to to leak up to 17725 bytes. This is quite large and likely due to a large heap allocation. The *memory sanitizer* (MSan) reported an uninitialized read when trying to leak 114 bytes or more. This number is probably a lower-bound as more bytes may be initialized when the server has been running for for a longer period of time.

Our framework reported a lot of diverged data when testing the known discriminator. This is because the `ssl3_write_bytes` function is called during split execution, which calls encryption and I/O logic, touching many data structures. Since *libdelta* dumps raw memory addresses and diverged data, which may differ per

Table 1: Number of discriminators (Discr), safe (S) and propagating (P) inputs found for each test server.

Server	Candidate	Discr.	P	S	Other
echo server	138	0	8	1	129
echo server (2)	30	1	9	9	11
replace server	265	79	0	90	96
quote server	278	1	210	67	0

execution, it is very challenging to determine what the diverged data represents, let alone whitelist it.

It turned out we could not test the fuzzer to find the known discriminator. This is because the heartbeat is encrypted and VUzzer taints bytes as soon as they are returned by `read()`, which is before decryption. Obviously, VUzzer is is unable to reason about the effects of a generated input on application logic executed after decryption. This is a general limitation of fuzzers that generally requires per-application knowledge to resolve.

5.2 Experiments

We have conducted a number of experiments with custom test servers, listed in Table 1. We show the number of candidate inputs generated by the fuzzer, followed by the number of discriminators found. The remaining candidate inputs are classified in *only-safe* (S), *only-propagating* (P) or *other*. *Echo server* can be found in Figure 2 and is completely different than *Echo server (2)* whose source listing is omitted for brevity, just as the other two test servers.

We have found at least one discriminator for three out of four servers. No discriminator was found for *Echo server* because the fuzzer is not aware of the memory layout, nor aware of which input bytes represent the *size* integer. *size* must be exactly 100 to find a discriminator.

In case of *Echo server (2)* the discriminator could be found. VUzzer analyzes *CMP* instructions and extracts their constant operand which is injected in the input at the matching byte-offset. This allowed VUzzer to execute the correct code path leading to the discriminator.

The *replace server* has many discriminators, but these are false positives. This is another example where *safe overflows* allow for discrimination. Unfortunately, the compiler instrumentation adds stack variables, making the *safe overflow zone* much larger than it is in non-instrumented binaries.

6 RELATED WORK

This work represents a new application of delta execution [10], which was originally devised for efficient on-line patch validation. Our implementation facilitates reliable detection of both state convergence and propagating inputs, whereas the original paper aims to support long-lived *multiple almost redundant executions (MAREs)*. The original implementation is different as it can merge immediately when finished executing the patch-site. It will retain both versions of the diverged state and will split again when any diverged state is accessed. Since our framework assumes rapid state convergence, we have omitted this feature.

ZMap [3] is a fast single packet network scanner, designed for internet-scale scanning. It can be used in conjunction with its sister project *ZGrab*, which enables stateful application-layer handshakes and is able to grab banners and detect vulnerability to Heartbleed. As opposed to our work, all its scanning capabilities are hand-coded so manual effort is required to support new vulnerabilities. Our work can thus be used to enhance projects such as *ZMap* and*ZGrab*, automatically providing payloads used for detecting the presence of vulnerabilities. We made use of the ZGrab codebase in our Heartbleed experiments.

Mutation testing [2, 4, 6, 11], also known as *mutant killing*, is a software testing technique that has some resemblance with discriminator discovery. To assess the effectiveness of a test-suite, hundreds of copies—the mutants—of a *program-under-test* are created. Each of these have a single-statement mutation, analogous to a software bug, which is created using a *mutation operator*. The number of mutants *killed* (discovered) by the test-suite gives a good indication of its coverage. In the context of our research, the patch would be a mutation and we need to generate a test to kill it. Tools exist which can kill mutants automatically, but these do not scale to support real-world software [2, 6, 11]. Also, mutation operators introduce simple and well-understood changes to a program, as opposed to security patches. Finally, the problem of finding *safe* inputs is not addressed.

7 DISCUSSION

The current implementation still lacks certain usability features. For instance, when the delta execution frameworks fails to merge its executions the output consists of raw data and memory addresses. While it might be possible to leverage debug information for some parts of the program, areas such as the heap still require more work to decipher.

We found that the effectiveness of the sanitizers to detect information leakage greatly depends on how the application manages its memory. We rely on uninitialized memory and fine grained heap allocations to accurately detect information disclosure. This is also based on the assumption that the information leak uses a `memcpy`'like operation to leak a contiguous range of memory.

Finally, this technique requires vulnerabilities that involve one or multiple attacker-controlled dimensions. In case of buffer overflows or information leakage through `memcpy()` an attacker typically controls the buffer length. Thus, these dimensions offer a search space in which discriminators can be found. Some vulnerability types, such as *null-pointer-dereference* bugs will not have safe discriminators because they are either triggered or not.

8 CONCLUSION

In this paper, we took the initial steps towards safe patch fingerprinting. It is a novel approach that could vastly improve the effectiveness of vulnerability scanners by automatically providing fingerprints for the most recent vulnerabilities. It supports detection of vulnerable servers through their behaviour, rather than relying on detection of the software version.

Delta execution turned out to be an effective technique to test whether an input is a discriminator. It allowed us to verify that

Heartbleed has a discriminator according to our definitions. However, whitelisting the benign diverged state was very time consuming. Demanding full state convergence is perhaps a bit too strict in real-world scenarios.

We modified an existing fuzzer to automate discriminator discovery. This lead to the discovery of discriminators in three out of four test servers. More work is needed to make automated discovery scale to realistic software.

This is work in progress. To determine how useful the scanning the technique is in practice, we should conduct more experiments with real-world applications. This is not trivial, as fully working patches and malicious seed inputs as required by our approach are not that widely available. Also, vulnerabilities whose patch does not introduce propagating inputs cannot be scanned. Exploiting side-channels and implementing delta execution on compiled binaries provide avenues to answer more thoroughly the question about the practicality of the approach.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing" and NWO 639.021.753 VENI "PantaRhei".

REFERENCES

[1] The Linux Kernel Archives. 2013. The Linux Kernel Archives: Soft Dirty PTE's. https://www.kernel.org/doc/Documentation/vm/soft-dirty.txt. (2013). Accessed: 2018-02-09.
[2] Richard A. DeMillo and A. Jefferson Offutt. 1991. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Softw. Eng.* (Sept. 1991).
[3] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2013. ZMap: Fast Internet-wide Scanning and Its Security Applications. In *USENIX Security*.
[4] William E. Howden. 1976. Reliability of the Path Analysis Testing Strategy. *IEEE Trans. Softw. Eng.* (May 1976).
[5] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*.
[6] Mike Papadakis and Nicos Malevris. 2010. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In *ISSRE*.
[7] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS*.
[8] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*.
[9] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: Fast Detector of Uninitialized Memory Use in C++. In *CGO*.
[10] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. 2009. Efficient Online Validation with Delta Execution. In *ASPLOS*.
[11] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. 2010. Test Generation via Dynamic Symbolic Execution for Mutation Testing. In *ICSM*.