Towards Deployment-Time Dynamic Analysis of Server Applications

Luís Pina   Cristian Cadar
Imperial College London
{l.pina, c.cadar}@imperial.ac.uk

Bug-finding tools based on *dynamic analysis (DA)* have become ubiquitous during software development. Notable examples include tools such as Valgrind [3] or the compiler sanitizers provided by Clang and GCC [5, 6]. While precise, these analyses incur a large performance overhead (often several times slower than native execution), which makes them prohibitively expensive to use in production. In current work, we are investigating the exciting possibility of deploying such dynamic analyses in production code, using a multi-version execution approach [1, 2].

Multi-version execution allows multiple program versions to run concurrently as long as they issue the same sequence of system calls. In particular, we have built a system called Varan [2] which resembles a record-replay system: a *leader* version performs the system calls and records their results into a buffer in shared memory, while other *follower* versions read the results of the system calls from the buffer, to mimic the leader's behaviour. System-call interposition is inexpensive, done through binary rewriting. For details on the architecture, we refer the reader to Hosek and Cadar [2]. The main advantage of this decentralised architecture is the small performance overhead, as the leader only needs to write the results of the system calls in the shared buffer, without having to synchronise directly with the followers.

To deploy a DA in a production system, we run a native version of the application as the leader, and a version instrumented for DA as the follower. Given that the buffer has a fixed size and the follower is much slower, the leader eventually runs out of free space in the buffer. Once this happens, the leader has to wait for the follower to consume the results in the buffer, effectively running at the same speed as the follower.

One solution for dealing with this problem is *partial DA checking*, i.e. only checking part of the execution with the DA. In this paper, we illustrate partial DA checking in the context of network servers and compiler sanitizers.

Case Study

For our study, we chose a toy web server implemented as part of a class assignment[1] which we further simplified, e.g., by limiting the number of connected clients to one. This allowed us to quickly experiment with our ideas, before moving to real-world applications.

Network servers are typically structured around a main dispatch loop which waits for user requests. As a result, user requests are the basic blocks of a server's execution. Running the DA on only a fraction of the requests is a natural way to do partial checking. To skip checking a request in our multi-version execution framework, the leader simply disables recording the system calls for that request. Since the web-server is stateless, the executions of the leader and follower do not diverge due to the leader handling more requests than the follower.

To detect each request, we added a simple annotation at the top of our server's dispatch loop, similar to the annotations done in the context of dynamic software updates [4]. More precisely, we added an *open* system call to the start of the loop with a unique dummy path as the argument. The dummy argument instructs Varan to ignore the system call, and also gives it the opportunity to detect the start of the loop and decide whether to send the system calls issued in that loop iteration to the follower.

For our dynamic analysis, we chose the Address Sanitizer (ASan) [5], a compiler sanitizer that detects buffer overflows and use-after-free errors in C/C++ programs. ASan is available in both Clang and GCC—for our experiments we used GCC 4.8.2.

The execution of the web server is dominated by I/O—reading the requested file and sending it to the client. As a result, the ASan overhead is relatively small (the requests/second rate drops to around 88% of the original rate). To mimic a server that also performs some computation for

CONF 'yy,   Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

[1] Abhijeet Rastogi's blog, `http://tinyurl.com/qxhezrb`

each request, thus introducing a significant sanitization overhead, we modified our web server to compress the requested files through BZip2 before sending them to the client.

Experimental Evaluation

To measure the web server's performance, we ran for 30s the *http_load* benchmark,[2] configured to download files in sequence, randomly chosen from a set of ten files with size 1KB and ten with size 1MB.

On average, an iteration of the server loop issues roughly 2K system calls. Therefore, we configured Varan to use a buffer with a capacity of 32K system calls, thus able to hold several loop iterations.

The top half of Figure 1 shows how Varan and ASan affect baseline performance. Each bar shows how many files were downloaded in 30 seconds. The bars labelled *No Varan* refer to running the web server without Varan, compiled with and without the sanitizer. We can see that ASan reduces throughput by roughly 50%. The bars labelled *Leader only* refer to repeating this experiment but running Varan with either a native or sanitized leader. We can see that Varan adds little overhead to the baseline, consistent with prior experiments [2]. Finally, the bars labelled *Leader/Follower* refer to Varan configured with a native leader and a native or sanitized follower. As expected, when a sanitized follower is used, the throughput drops by almost half, because once the buffer gets full, the leader has to wait for the slow follower.

To throttle sending system calls to the follower, Varan only sends the first $n\%$ in each window of 32K system calls. We configured Varan to repeat this process with an increasing n, from 0 to 100 in increments of 10. Given that Varan has to wait for the start of the loop, the actual percentage varies. The bottom half of Figure 1 shows the results for this experiment. The label on each bar describes how many requests Varan checked in the sanitized follower.[3] We can see that the results match expectations, with throughput increasing as the percentage of checked requests decreases.

Related Work

We briefly introduced the idea of production-time sanitization in our Varan paper [2] and illustrated its applicability in the context of an I/O-bound server, where the sanitized follower can easily keep up with the leader. In this work, we focus instead on the more challenging scenario in which the sanitized follower cannot.

The research community has invested significant effort into designing DA techniques that are cheap enough to be deployed in production. Instead, our focus is to make *existing* expensive techniques, such as compiler sanitizers, deployable in production.

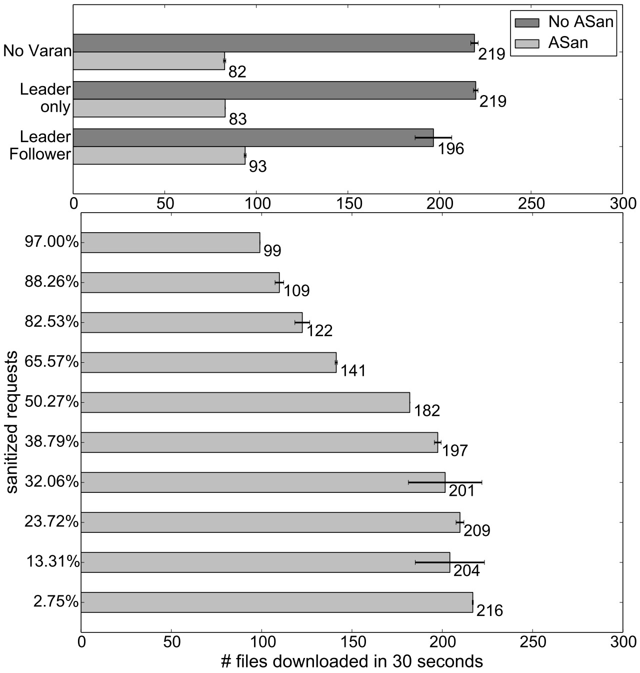

Figure 1. Web-server performance experiment. Each bar reports the average throughput over 10 runs, with the error bars reporting the standard deviation. Run on a an Intel Xeon E31280 machine, 3.5 GHz CPU (8 logical cores, 4 physical), 16GB of RAM, with Ubuntu Linux 14.04 (kernel 3.13.0-39).

Conclusion

Our preliminary experience makes us optimistic that partial checking of network server executions during deployment is feasible. In future work, we plan to focus on the challenges introduced by real-world servers, as well as explore new application domains.

Acknowledgements. We thank Petr Hosek for his assistance with the Varan system, and the EPSRC for sponsoring this research through an Early-Career Fellowship.

References

[1] P. Hosek and C. Cadar. Safe software updates via multi-version execution. In *ICSE'13*.

[2] P. Hosek and C. Cadar. Varan the Unbelievable: An efficient N-version execution framework. In *ASPLOS'15*.

[3] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI'07*.

[4] L. Pina, L. Veiga, and M. Hicks. Rubah: DSU for Java on a stock JVM. In *OOPSLA'14*.

[5] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC'12*.

[6] E. Stepanov and K. Serebryany. MemorySanitizer: fast detector of uninitialized memory use in C++. In *CGO'15*.

[2] `http://www.acme.com/software/http_load`

[3] The percentage of system calls replayed matches the checked requests.