EmbedSanitizer: Runtime Race Detection Tool for 32-bit Embedded ARM

Hassan Salehe Matar, Serdar Tasiran and Didem Unat

Koç University, Istanbul, Turkey
{hmatar, stasiran, dunat}@ku.edu.tr

Abstract. We propose EmbedSanitizer, a tool for detecting concurrency data races in 32-bit ARM-based multithreaded C/C++ applications. Moreover, we motivate the idea of detecting data races in embedded systems software natively; without virtualization or emulation or use of alternative architecture. Detecting data races in applications on a target hardware provides more precise results and increased throughput and hence enhanced developer productivity. EmbedSanitizer extends ThreadSanitizer, a race detection tool for 64-bit applications, to do race detection for 32-bit ARM applications. We evaluate EmbedSanitizer using PARSEC benchmarks on an ARMv7 CPU with 4 logical cores and 933MB of RAM. Our race detection results precisely match with results when the same benchmarks run on 64-bit machine using ThreadSanitizer. Moreover, the performance overhead of EmbedSanitizer is relatively low as compared to running race detection on an emulator, which is a common platform for embedded software development.

1 Introduction

Embedded systems are everywhere: from TVs to robots to smartphones to Internet of Things. Moreover, the computing capability of these systems has tremendously increased in recent years due to multicore support. This has enabled the implementation of complex multithreaded parallel applications. Unfortunately, these applications are prone to concurrency errors such as data races. These bugs are hard to detect in nature and the availability of relevant tools for embedded systems is still limited.

Most of the software development environment for Embedded systems rely on hardware emulations, which tend to be slow. Race detection of embedded system software through emulation can add even more overhead. Nevertheless, running software for race detection on a real hardware not only provides precise race reports but also is faster and hence more productive.

Moreover, many practical race detection tools for C++ applications have not focused on embedded system architectures. Therefore, the alternative is to compile 32-bit embedded C++ applications for other architectures and do race detection there. Unfortunately, some parts of the software that use special features of the target hardware may not be checked due to unavailability of such features in alternative platforms. Further, it is more appealing to use full features of the software on the target devices for race detection.

We propose a tool named *EmbedSanitizer* [1] for detecting data races for multithreaded 32-bit Embedded ARM software at runtime by running the instrumented application in the target platform. There are two advantages of this approach: (a) parts of software which use unique features, like sensors and actuators, can be analyzed. (b) enhanced developer productivity and throughput attained due to increased performance of race detection compared to hardware emulator. Our tool modifies *ThreadSanitizer* [18] to support race detection for the embedded ARMv7 architecture. Moreover, LLVM/Clang is modified to support *EmbedSanitizer* so that it launches in a similar manner to *ThreadSanitizer*. For simplicity, *EmbedSanitizer* has an automated script which downloads necessary components and builds them together with LLVM/Clang as a cross-compiler. Multithreaded C/C++ programs through this compiler are instrumented and finally run on the target 32-bit ARM hardware for race detection.

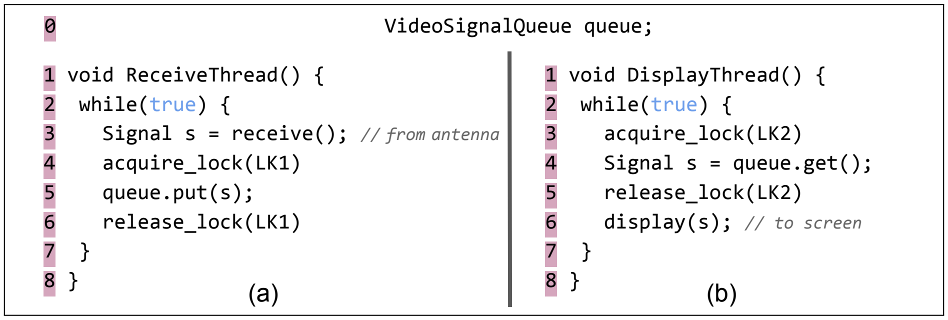
The key contributions of this paper are as following:

1. We present a tool for detecting data races in C/C++ multithreaded programs for 32-bit embedded ARM. The tool is easily accessed through Clang compiler chain like *ThreadSanitizer*.
2. We motivate the idea of supporting race detection in native embedded systems hardware and show usability of race detection on such architectures.
3. We evaluate our tool and show its applicability by running PARSEC benchmark applications on a TV with ARMv7 CPU.

2 Motivation

We aim to promote utilization of existing race detection tools by adapting them to different hardware architectures. To show benefits of this approach, consider a theoretical multithreaded example in Figure 1. It models a TV software component which has two concurrent threads. `ReceiveThread` reads TV signals from an antenna and puts data in a shared queue *queue*. Then `DisplayThread` removes the data from the queue and displays on the TV screen. For the sake of motivation, the implementation of the queue is abstracted away but uses no synchronization to protect concurrent accesses. Since `ReceiveThread` and `DisplayThread` do not use a common a lock (`LK1 & LK2` are used) to protect accesses to *queue*, there is a data race at lines 5(a) and 4(b).

Assume that the developer chooses a method other than the proposed one for race detection. She has two challenges: (1) Modeling the *receipt* as well as the *display* of the video signal data. (2) After that, she can do race detection on an alternative architecture, emulation or virtualization rather than the target architecture. Further overhead is incurred if emulation or virtualization is used. Conversely, the target hardware already has these features and may be faster and thus increasing developer productivity. Moreover, the advantage of instrumenting program and later detecting races on a target hardware is that



```
                    VideoSignalQueue queue;
```

```
void ReceiveThread() {
 while(true) {
   Signal s = receive(); // from antenna
   acquire_lock(LK1)
   queue.put(s);
   release_lock(LK1)
 }
}
```

(a)

```
void DisplayThread() {
 while(true) {
   acquire_lock(LK2)
   Signal s = queue.get();
   release_lock(LK2)
   display(s); // to screen
 }
}
```

(b)

Fig. 1. A motivating example with two threads concurrently accessing a shared queue. A thread in (a) reads video signals from TV antenna and puts them into the queue, (b) reads from the queue and display to a screen.

the developer uses real features for receiving and displaying the signals. This aligns exactly well with our proposed solution.

3 Related Work

Zeus Virtual Machine® Dynamic Framework [22, 21] is a hardware-agnostic platform which contains tools for detecting runtime data races for kernel and user-space multithreaded applications. These tools rely on virtualization and may abstract away real target system interactions with external peripherals like sensors. Moreover, these tools are proprietary and not much relevant information is in the literature. Conversely, *EmbedSanitizer* is open-source and does not rely on virtualization. Differently, from these tools, we motivate the use of the real target hardware for race detection. This improves runtime performance with high precision and developer productivity.

Most of the related solutions for detecting data races do target low end interrupt based, non-multithreaded embedded systems [19, 20, 6, 23]. Therefore, these solutions can not be directly applied to the multithreaded software for ARMv7. Moreover, Keul [12] and Chen [7] use static analysis techniques for race detection in interrupt-driven systems applications. Unfortunately, these techniques do not capture the runtime behavior of the program. Therefore, they fail to infer many of execution patterns which would otherwise result in data races.

Goldilocks [8] is a framework for Java programs which triggers an exception when a race is about to happen. It uses lockset-based and happens-before approaches to improve precision of race detection as well as static analysis to filter out local memory accesses for improved runtime overhead. Differently from our approach, Goldilocks targets Java programs and needs Java virtual machine which may not be ideal for embedded systems.

Finally, *Intel Inspector XE* [11], *Valgrind DRD* [14, 3] and *ThreadSanitizer* [18] are race detection tools for C/C++ multithreaded programs. Despite running on native hardware, these tools have limited support for 32-bit ARM architectures. Therefore, they can not directly be used for ARMv7 [4]. *ThreadSanitizer* [18], for example, is developed by default for x86_64 architecture.

4 Background

This section discusses various concepts employed in *EmbedSanitizer*.

ThreadSanitizer. *ThreadSanitizer* [18] is an industrial-level and open-source race detection tool for Go, and C/C++ for 64-bit architectures. This tool is accessible through GCC and LLVM/Clang [13] using compiler flag `-fsanitize=thread`. It instruments the program under compilation by identifying shared memory and synchronization operations and injecting runtime callbacks. The instrumented executable is then run on a target platform for detecting races.

ThreadSanitizer has been successful mainly for two reasons. First, it uses a hybrid of *happens-before* and *lockset* algorithms to improve its precision. Second, it uses 64-bit architectural capability to store race detection meta-data called *shadow memory* for performance and memory efficiency. The authors of *ThreadSanitizer* claim that extending it for 32-bit applications is *unreliable and problematic* [2]. Therefore, we benefit from its instrumentation part and extend it to support race detection of 32-bit ARM applications.

Data Races. A data race [15], [17] occurs when two concurrently executing threads access a shared memory location without proper synchronization and at least one of these accesses is a write. Availability of data races in a program can be a symptom of higher concurrency errors such as atomicity and linearizability violations, and deadlocks. Moreover, data races may result in non-deterministic behavior like memory violations and program crashes.

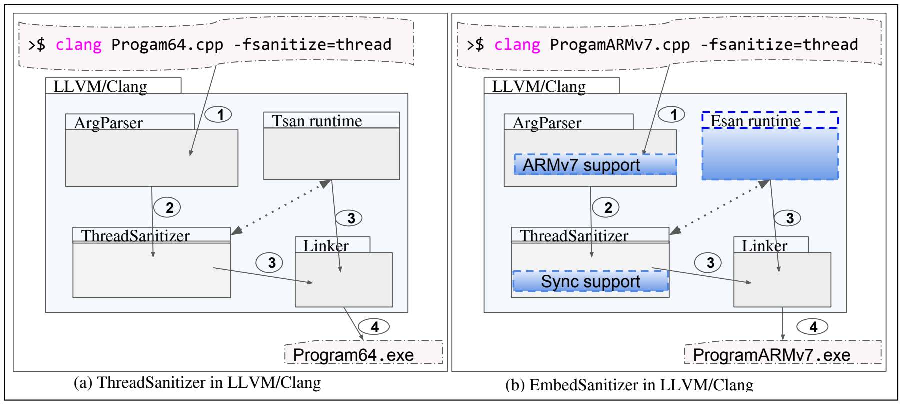
FastTrack Race Detection Algorithm. FastTrack [9] is an efficient and precise race detection algorithm which improves on purely happens-before vector clock algorithms such as DJIT++ [16]. FastTrack shows that majority of memory access patterns do not require a whole vector clock to detect data races. Instead, an *epoch*, a simple pair of thread identifier and clock suffices. Without sacrificing precision, this significantly improves the performance of race detection of a single memory access from O(n) to O(1) where n is the number of concurrent threads in the program under test. Moreover, its runtime performance is better than most of the race detection algorithms in the literature [24]. Finally, there are further improvements to FastTrack algorithm but tend to sacrifice precision [10].

5 Methodology

EmbedSanitizer improves on *ThreadSanitizer*. It can also be launched through Clang's compiler flag `-fsanitize=thread`. To achieve this, we modified the LLVM/Clang compiler argument parser to support instrumentation of 32-bit ARM programs when the relevant flag is supplied at compile time. Next, *EmbedSanitizer* enhances parts of the *ThreadSanitizer* to instrument the target program. Furthermore, it replaces the 64-bit race detection runtime with a custom implementation of the efficient and precise FastTrack race detection algorithm, for 32-bit platforms. In this section, we discuss the important parts of *EmbedSanitizer* as well as its simplified installation process.

5.1 Architecture and Workflow

Workflow of the *ThreadSanitizer* and the changes done by *EmbedSanitizer* are described in Figure 2. Figure 2(a) shows default and unmodified relevant components of *ThreadSanitizer* in LLVM/Clang. In Figure2(b) these parts are modified to enable instrumentation and detection of races for 32-bit ARM applications.



```
>$ clang Progam64.cpp -fsanitize=thread
```

```
>$ clang ProgamARMv7.cpp -fsanitize=thread
```

(a) ThreadSanitizer in LLVM/Clang

(b) EmbedSanitizer in LLVM/Clang

Fig. 2. High level abstraction of *ThreadSanitizer* and *EmbedSanitizer* in LLVM/Clang. In (a) *ThreadSanitizer*: essential LLVM modules for race detection. In (b) *EmbedSanitizer*: same modules modified to instrument and detect races for 32-bit ARM

At ① in Figure 2(a), the Clang front-end reads the compiler arguments and parses them. If the target architecture is 64-bit, Clang passes the program under compilation through *ThreadSanitizer* compiler pass for instrumentation ②. The pass then identifies all shared memory operations in the program and injects relevant race detection callbacks which are implemented in a race detection runtime library called *tsan*. Furthermore, the instrumented application and the runtime are linked together by the linker ③ to produce an instrumented executable④. This executable once runs on a target 64-bit platform, it reports race warning in the program. We modify components in the workflow as discussed next.

(a) Enabling Instrumentation of 32-bit ARM Code in LLVM/Clang: We modify the argument parser of LLVM/Clang to support instrumentation once *EmbedSanitizer* is in place, Figure 2(b). Therefore, if `-fsanitize=thread` flag is passed while compiling a program for 32-bit ARM code, the instrumentation takes place. To do this we identified the locations where Clang processes the flag and checks the hardware before skipping the launching of *ThreadSanitizer* instrumentation module because of unsupported architecture.

(b) Modifying the *ThreadSanitizer* Instrumentation Pass: Despite its instrumentation pass, *ThreadSanitizer* has become complex, partly due to its integration into the LLVM's compiler runtime. We extended the available instrumentation pass to identify and instrument synchronization events and inject relevant callbacks and kept instrumentation of memory accesses as it is.

(c) Implementation of Race Detection Runtime: The default race detection runtime in *ThreadSanitizer* uses memory shadow structures which rely on 64-bit architectural support. Due to the complicated structure of ThreadSanitizer, it was not possible to adopt its runtime for 32-bit ARM platform. Therefore, we implemented a race detection runtime by applying the FastTrack race detection algorithm. The library is then compiled for 32-bit ARM and is linked to the final executable of the embedded program at compile time.

5.2 Installation

Figure 3 shows the building process of LLVM compiler infrastructure with *EmbedSanitizer* support. To simplify this process we developed an automated script with five steps. In the first step, it downloads the LLVM source code from the remote repository. Then it replaces files of the LLVM/Clang compiler argument (flags) parser with our modified code to enable *ThreadSanitizer* support for ARMv7. Third, the LLVM code is compiled using GNU tools to produce a cross-compiler which targets 32-bit ARM and supports our tool, *EmbedSanitizer*. Fourth, the race detection runtime which we implemented is compiled separately and integrated into the built cross-compiler binary. Finally, the built cross-compiler is installed which can eventually be used to compile 32-bit ARM applications with race detection support. This whole process is applied once.

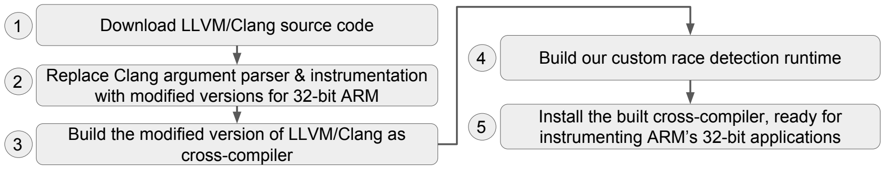

1
Download LLVM/Clang source code
2
Replace Clang argument parser & instrumentation with modified versions for 32-bit ARM
3
Build the modified version of LLVM/Clang as cross-compiler
4
Build our custom race detection runtime
5
Install the built cross-compiler, ready for instrumenting ARM's 32-bit applications

Fig. 3. Showing the automated process for building *ThreadSanitizer* for the first time.

6 Evaluation

We evaluate *EmbedSanitizer* for detecting runtime data races for 32-bit embedded ARM applications, based on two categories. First, we want to see how the precision of race detection in *EmbedSanitizer* deviates from that of *ThreadSanitizer* [18] since *EmbedSanitizer* extends it by using its instrumentation features, and implements a custom FastTrack [9] for detecting races. Second, we want to compare the overhead of *EmbedSanitizer* when running on a target embedded device against when running on an emulator. The key motivation is to show that running race detection on a target device is better than on emulation.

For experimental setup, we built LLVM/Clang, with *EmbedSanitizer* tool, as a cross-compiler in a development machine running Ubuntu 16.04 LTS with Intel i7 (x86_64) CPU and 8GB of RAM. As our benchmarks, we picked four(4) of the PARSEC benchmark [5] applications. We adopted these applications to Clang compiler and our embedded system architecture. A short summary about the applications we used for evaluation is given below.

- *Blackscholes:* parallelizes the calculation of pricing options of assets using the Black-Scholes differential equation.
- *Fluidanimate:* uses spatial partitioning to parallelize the simulation of fluid flows which are modeled by the Navier-Stokes equations using the renowned Smoothed particle hydrodynamics.
- *Streamcluster:* is a data-mining application which solves the k-means clustering problem.
- *Swaptions:* employs Heath-Jarrow-Morton framework with Monte Carlo simulation to compute the price of a set of swaptions.

6.1 Tool Precision Evaluation

We compare the race reports detected by *EmbedSanitizer* against *ThreadSanitizer*. To do this we run the same benchmark applications with *ThreadSanitizer*, as well as with *EmbedSanitizer*. The instrumented program using *ThreadSanitizer* is run on an x86_64 machine, whereas the binary compiled through *EmbedSanitizer* is executed on ARM Cortex A17 TV. In this setting of four PARSEC benchmark applications, in an application where *ThreadSanitizer* reported races, *EmbedSanitizer* also reported them as shown in Table 1. Therefore *EmbedSanitizer* did not sacrifice any race detection precision.

Table 1. Experimental results to compare race detection in ARMv7 using *EmbedSanitizer* vs in x86_64 with *ThreadSanitizer*.

	Input						*ThreadSanitizer*	*EmbedSanitizer*
Benchmark	**size**	Threads	Addresses	Reads	Writes	Locks	Races	Races
blackscholes	4K options	2+1	28686	5324630	409590	0	NO	NO
fluidanimate	5K particles	2+1	149711	25832663	8457516	790	YES	YES
streamcluster	512 points	2+1	11752	21710589	352605	2	YES	YES
swaptions	400 simulations	2+1	243945	11000763	3377226	0	NO	NO

6.2 Tool Performance Evaluation

To compare race detection overhead, we ran non-instrumented and instrumented versions of the benchmarks on embedded TV with ARM-Cortex A17 CPUs of 4 logic cores and 933MB of RAM, and on Qemu-ARM emulator running on a workstation. The slowdown is calculated as a ratio of the execution time of the instrumented program with race detection on and the execution time of the program without race detection. The number of threads was 3 because using the full set of 4 logical cores was crashing the TV. Next, the input sizes were the same in each benchmark setting. Results in Figure 4 show that detecting races in an emulator incurs between 13x and 371x slowdown whereas the slowdown in the TV is between 12x and 214x. In overall, results in Figure 4 suggest that detecting races in a target hardware is faster than in an emulator.

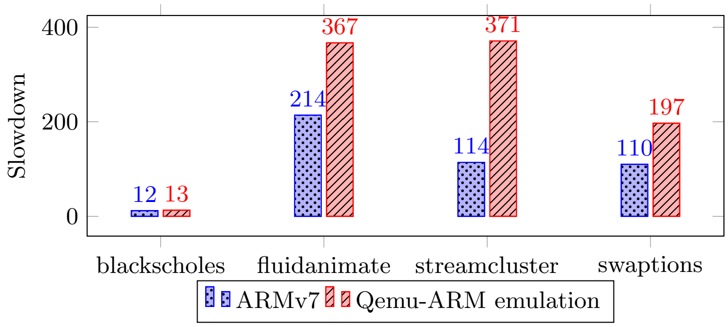


Fig. 4. Slowdown comparison of race detection on ARMv7 vs on Qemu-ARM

7 Conclusion and Future Work

This paper presented *EmbedSanitizer*, a tool for detecting data races for applications targeting 32-bit ARM architecture. *EmbedSanitizer* extends *ThreadSanitizer*, a race detection tool widely accessible through Clang and GCC, by enhancing its instrumentation. Moreover, we implemented our own 32-bit version of race detection runtime to replace *ThreadSanitizer*'s race detection runtime which is incompatible with 32-bit ARM. Our custom race detection library adopts FastTrack, an efficient and precise happens-before based algorithm.

To evaluate the consistency of *EmbedSanitizer*, we used four PARSEC benchmark applications. First, we evaluated the precision of the tool by comparing the race report behavior with that of *ThreadSanitizer*. Next, we compared its slowdown with running race detection on the Qemu emulator as a representative for testing ARM code in a high-end developer platform.

As a future work, there are four areas to improve. First, improving the efficiency of the custom race detection runtime by hybridizing it with other race detection algorithms. Second, supporting other 32-bit based architectures like the Intel's IA-32. Third, evaluating *EmbedSanitizer* with real-world applications which use special features of the embedded systems such as sensors and actuators, which is the real motivation of our work.

Acknowledgements: This work has been funded under the Affordable Safe & Secure Mobility Evolution (ASSUME) project for smart mobility. We also thank Arçelik A.Ş. for providing the platforms to evaluate our method.

References

1. Embedsanitizer, https://www.github.com/hassansalehe/embedsanitizer
2. Threadsanitizer documentation, https://clang.llvm.org/docs/ThreadSanitizer.html
3. Valgrind drd (2017), http://valgrind.org/docs/manual/drd-manual.html
4. ARM: Arm architecture reference manual armv7-a and armv7-r edition issue c, https://silver.arm.com/download/download.tm?pv=1603196
5. Bienia, C.: Benchmarking Modern Multiprocessors. Ph.D. thesis, Princeton University (January 2011)
6. Chen, R., Guo, X., Duan, Y., Gu, B., Yang, M.: Static data race detection for interrupt-driven embedded software. In: 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement - Companion. pp. 47–52 (June 2011)
7. Chen, R., Guo, X., Duan, Y., Gu, B., Yang, M.: Static data race detection for interrupt-driven embedded software. In: 2011 Fifth International Conference on Secure Software Integration and Reliability Improvement - Companion. pp. 47–52 (June 2011)
8. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: A race and transaction-aware java runtime. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 245–255. PLDI '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1250734.1250762
9. Flanagan, C., Freund, S.N.: Fasttrack: Efficient and precise dynamic race detection. In: Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 121–133. PLDI '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1542476.1542490
10. Ha, O.K., Jun, Y.K.: An efficient algorithm for on-the-fly data race detection using an epoch-based technique. Sci. Program. 2015, 13:13–13:13 (Jan 2015), https://doi.org/10.1155/2015/205827
11. Intel: Intel inspector xe (2017), https://software.intel.com/en-us/intel-inspector-xe
12. Keul, S.: Tuning static data race analysis for automotive control software. In: 2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation. pp. 45–54 (Sept 2011)
13. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (Mar 2004)
14. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Not. 42(6), 89–100 (Jun 2007), http://doi.acm.org/10.1145/1273442.1250746
15. Netzer, R.H.B., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst. 1(1), 74–88 (Mar 1992), http://doi.acm.org/10.1145/130616.130623
16. Pozniansky, E., Schuster, A.: Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. Concurrency and Computation: Practice & Experience 19(3), 327–340 (2007)
17. Qadeer, S., Tasiran, S.: Runtime verification of concurrency-specific correctness criteria. International Journal on Software Tools for Technology Transfer 14(3), 291–305 (2012), http://dx.doi.org/10.1007/s10009-011-0210-1
18. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: Data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. pp. 62–71. WBIA '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1791194.1791203

19. Tchamgoue, G.M., Kim, K.H., Jun, Y.K.: Dynamic Race Detection Techniques for Interrupt-Driven Programs, pp. 148–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2012), http://dx.doi.org/10.1007/978-3-642-35585-1_20
20. Tchamgoue, G.M., Kim, K.H., Jun, Y.K.: Verification of data races in concurrent interrupt handlers. International Journal of Distributed Sensor Networks 9(11), 953–593 (2013), http://dx.doi.org/10.1155/2013/953593
21. Wire, B.: Parallocity licenses zeus virtual machine dynamic analysis framework to h3c technologies. http://www.businesswire.com/news/home/20121211005482/en/Parallocity-Licenses-Zeus-Virtual-Machine%C2%AE-Dynamic-Analysis (2012)
22. Wire, B.: Akamai selects parallocity's zvm-u dynamic software analysis framework. http://www.businesswire.com/news/home/20130305005107/en/Akamai-Selects-Parallocity%E2%80%99s-ZVM-U-Dynamic-Software-Analysis (2013)
23. Wu, X., Wen, Y., Chen, L., Dong, W., Wang, J.: Data race detection for interrupt-driven programs via bounded model checking. In: 2013 IEEE Seventh International Conference on Software Security and Reliability Companion. pp. 204–210 (June 2013)
24. Yu, M., Park, S.M., Chun, I., Bae, D.H.: Experimental performance comparison of dynamic data race detection techniques. ETRI Journal, vol. 39, no. 1, Feb. 2017 39(1), 124–134 (Feb 2017)