Enhancing S2E to Analyze Multi-Thread Programs

Fedor Niskov
MSU, ISP RAS
Moscow, Russia
e-mail: fedor.niskov@ispras.ru

Egor Kutovoy
MIPT
Moscow, Russia
e-mail: kutovoi.ea@phystech.edu

Shamil Kurmangaleev
ISP RAS
Moscow, Russia
e-mail: kursh@ispras.ru

Abstract — **Code analysis for defect detection is very important in the modern world, especially in the case of complex multi-thread applications. An example of a tool, suitable for software of high complexity, is the famous S2E, which allows for full-system emulation with symbolic execution. This paper presents several major enhancements for S2E, including: firstly, support for multiple virtual cores, allowing to have parallel speed-up; secondly, on this basis, a race checker plugin to detect defects of this sort in multi-thread programs. This development has concerned such interesting points of research as scheduling in multi-core emulation and race detection with symbolic execution.**

Keywords — **S2E, full-system emulation, symbolic execution, multi-threading, race detection, parallel speed-up.**

I. Introduction

Code analysis is a very important research direction nowadays. Special analysis tools are required to find defects in complex modern software. Today developers design their programs to leverage parallel capabilities of multi-core processors, so multi-thread programs are quite widespread, and they also need to be analyzed – such heisenbugs as *races* are hard for manual debugging and need special attention.

Many issues arise in code analysis. An important aspect is making a proper environment for the target program. A possible solution here is full-system emulation: not only the target software is emulated, but also all related components, including the operating system. A prominent example in this area is the S2E [1][2] platform, allowing for symbolic execution based on full-system emulation. Despite the project's definite success, there is a room for improvements and new features.

This paper presents the results of our work – a set of improvements to enhance S2E, which can be summarized as two main achievements:

- Support for multiple virtual cores in parallel threads.
- Implementation of a race checker in the multi-core emulator.

The following sections reveal the details of this work.

II. Overview of S2E

First of all, it's worth describing the general design of S2E [1][2]. This is a platform for full-system emulation with symbolic execution. The name means "Selective Symbolic Execution", i.e., some code is executed concretely, and some – symbolically. S2E allows to make interesting bytes symbolic, and to traverse the tree of possible states, i.e., to explore the branches which these bytes influence. As a result, a collection of input data variants is generated for code coverage.

S2E is based on the Qemu [3] emulator – it is launched in the KVM mode, but the original KVM module is replaced with a special component (`libs2e*.so`, via `LD_PRELOAD` and intercepting `ioctl`). The S2E library implements the KVM interface – so it is responsible for emulation of the processor and memory, while Qemu emulates the peripheral devices and controls the whole process. The library uses a modified TCG engine from Qemu for concrete execution and the KLEE [4] framework for symbolic execution. The original Qemu (v3.0) has been patched for integration with S2E.

The S2E library is compiled in three forms, which can be briefly described as pre-snapshot, concrete, and symbolic execution. The first form serves for limited concrete execution (with many S2E features disabled) – to boot the virtual machine and save a snapshot. The latter forms are used for execution after loading the snapshot.

S2E is highly extensible – there is a number of built-in plugins, and the user can write his own plugin in C++, on top of the basic API.

Thus, S2E is an advanced complex technology. Let's emphasize several aspects subject to improvement. The original S2E supports for only one virtual core – it precludes parallel speed-up in multi-thread programs and influences hardware-sensitive software. The Qemu component is based on the old Qemu v3.0 – an upgrade can be favorable for further development, including emulation of modern peripherals. Finally, S2E provides a fine platform for extra analysis checkers, and it's convenient for implementation of a new checker, aimed at race bugs in multi-thread programs.

III. Support for Multiple Cores

The enhanced S2E supports for multiple virtual cores. In brief, changes in the code for this purpose can be outlined as follows:

- Support for multiple objects of the VCPU class (representing virtual cores) – each of them acts in a separate thread.

https://doi.org/10.51408/csit2023_16

- Several related global structures are made private per-core; access to shared structures is protected.
- An execution state includes multiple core states (sets of registers).
- Hybrid inter-core synchronization, combining parallel and in-turn execution; a special scheduler controls cores:
  - When AP-cores* execute the target code in the concrete mode, they work simultaneously.
  - Otherwise, cores are serialized by the scheduler.
- Other changes for consistency.

* *Note: BSP-core – the core with ID=0, AP-core – a core with ID≠0.*

The innovation of such multi-core scheduler should be remarked, as opposed to the ordinary emulators, having only a simple round-robin single-thread scheduler or an uncontrollable form of multi-threading. The suggested scheduler allows to gain the profits of parallelism, while having a subtle control over cores – it can be beneficial for analysis and debugging.

There is an important note about the development: as mentioned above, the original Qemu component is based on the old version v3.0; during this work, it has been upgraded to v6.1. This modernization lets us use the advantages of the newer Qemu, laying a solid foundation for our work on parallelism and other directions.

This implementation has been tested on real software – Suricata [5] – a multi-thread program, acting as an IDS/IPS (Intrusion Detection/Prevention System). The following testcase is considered: Suricata handles a large bunch of packets with concrete data in parallel threads, then the final packets with symbolic data. The experiment involves an unmodified version of Suricata with a custom plugin (packet-checking handler), which allows to achieve a high workload for efficient parallelism. The system is configured to ensure a proper testing environment and conditions for parallelism in the aforesaid hybrid scheme.

The testing is successful: increasing the number of threads makes it faster, showing a significant parallel speed-up, and the symbolic functionality is correct – new inputs are generated for code coverage. The speed-up diagram is shown in Fig. 1 (up to 100 threads). The non-linearity has the following reasons: the limitations of the hybrid scheme; packet distribution is charged to a single thread; the distribution is not completely balanced (some threads get more packets than others); non-parallel code spoils speed-up according to Amdahl's law [6].

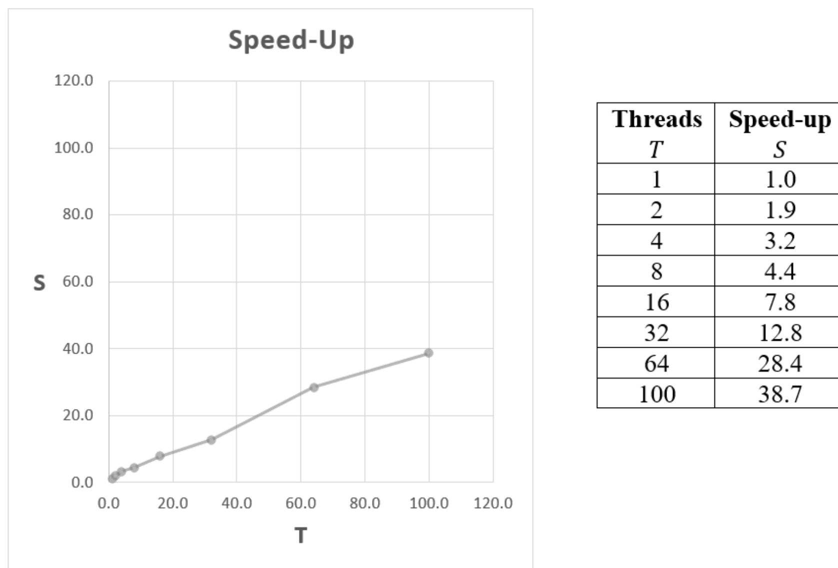
Thus, the testing has proven vitality of the suggested solution. The enhanced S2E can gain the powers of modern machines for high performance on many cores, and such multi-core emulation can trigger new behaviours in multi-thread applications, that can facilitate their analysis for detection of concurrency defects, such as races.

IV. Race Checker

Before describing our race checker for S2E, several general notes should be made about the field of race detection. As for the terminology in this paper, a *(data) race* means an unsynchronized multi-thread access to the same shared variable with writing. According to the classic approach, a race checker should monitor shared variable access and mutex operations in order to detect an unordered access. Typically, it implies a certain formalization with order relations and special clocks, such as vector clock, logical clock, Lamport clock [7], etc. Existing algorithms can be classified by the basic concept: happens-before-based (DJIT^{+} [8], FastTrack [9], LiteRace [10], LOFT[11]), lock-set-based (Eraser [12], Goldilock [13], paper [14]), and hybrid (AccuLock [15], RaceTrack [16], Helgrind^{+} [17]). The most eminent tools for race detection include Sanitizers [18] (TSan [19]) and Valgrind [20] (Helgrind [21], DRD [22]). There are also other related tasks in this area [23][24]: deterministic multi-threading, record and replay, execution synthesis, etc. Remarkable tools include CLAP [25], Symbiosis [26], Cortex [27], ESD [28], ODR [29], Tern [30].

Our race checker follows the aforementioned classic approach; it is mainly inspired by the DJIT^{+} [8] algorithm. Firstly, let's formalize the notion of a race.

As a prerequisite, the concept of *happens-before* (HB) is needed. This is a strict partial order relation on the set of the program's events (E). Let E_T^t be the event of command



Threads *T*	Speed-up *S*
1	1.0
2	1.9
4	3.2
8	4.4
16	7.8
32	12.8
64	28.4
100	38.7

Fig. 1: The speed-up diagram for Suricata

execution in thread T at time t. The happens-before relation is determined by the following rules:

- order in the same thread:
  $t1 < t2 \Rightarrow E_T^{t1} \prec_{HB} E_T^{t2}$
- order between threads via a mutex:
  $t1 < t2 \ \& \ unlock_{Mi}(E_{T1}^{t1}) \ \& \ lock_{Mi}(E_{T2}^{t2})$
  $\Rightarrow E_{T1}^{t1} \prec_{HB} E_{T2}^{t2}$

Having this background, it's possible to give a formal definition of a *(data) race*:

$$\textbf{Race} \Leftrightarrow \exists\, a, b \in E, \textit{var}\ v : \\ Access_v(a) \ \& \ Access_v(b) \ \& \ (Write(a) \ || \ Write(b)) \\ \& \ \neg\, (HB(a, b) \ || \ HB(b, a))$$

To track the happens-before relation in practice, the concept of *vector clock* is used – an array of time counters, each of them corresponding to a certain thread. Notably, vector timestamps can be viewed as a join-semilattice (regarding element-wise maximum and comparison).

For each shared variable, the checker keeps information about the last reading and writing (the accessing thread and the timestamp); each mutex has an associated timestamp; each thread has its own vector clock instance. The checker intercepts access to shared variables and mutex operations, updating the timestamps accordingly. If some access has a conflict with a previous one (as per the definition above: the same variable, at least one writing, the timestamps are not ordered), then the checker reports a race alarm.

According to the general design for such tools, the checker is implemented as an S2E plugin.

This detection pairs well with symbolic execution: when the symbolic engine traverses the tree of states, the checker performs this analysis along the flow of each state, so the engine can generate new inputs, including race-causing ones, and the checker can detect the race there.

According to our experience, the usage of symbolic features is a promising research direction. We augment the capabilities of race detection, involving the symbolic engine and building it all as an integral complex. Such tool is able to achieve higher results than the original algorithm.

The race checker was also tested on Suricata. An artificial race bug was inserted into the code: with certain values of bytes in the input packets, a shared variable was incremented without mutex protection. It incurred a race between packet-handling threads, running on multiple virtual cores. Eventually, the tool successfully found the race – the input with the special values was generated and the checker detected the race in this state.

V. Conclusion

Thus, two major improvements have been made to enhance S2E. Firstly, the emulator now supports multiple virtual cores, running in parallel threads – it allows to emulate multi-thread programs with parallel speed-up. This work has entailed vast changes in the codebase, including support for multi-threading and modernization of related components. An important research result here is the multi-core scheduler, allowing to have advanced patterns of inter-core synchronization. Secondly, a race checker has been implemented on the base of S2E, and it can be useful to find defects in multi-thread programs. Combination of symbolic execution and race detection is a perspective research direction. In total, our testing experience has shown many positive effects of this enhancement, and inspires us for new improvements.

References

[1] (2023) The S2E website. [Online]. Available: https://s2e.systems

[2] V. Chipounov, V. Kuznetsov, G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems", *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 265–278, 2011.

[3] F. Bellard, "QEMU, a fast and portable dynamic translator", *Proceedings of the USENIX Annual Technical Conference (ATEC'05)*, Berkeley, CA, USA, pp. 41–46, 2005.

[4] C. Cadar, D. Dunbar, D.R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs", *OSDI*, vol. 8, pp. 209–224, 2008.

[5] (2023) The Suricata website. [Online]. Available: https://suricata.io

[6] G.M. Amdahl, "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", *AFIPS Conference Proceedings*, vol. 30, pp. 483–485, 1967.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system", *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[8] E. Pozniansky, A. Schuster, "Efficient on-the-fly Data Race Detection in Multithreaded C++ Programs", *PPoPP'03*, pp. 179–190, 2003.

[9] C. Flanagan, S. N. Freund, "FastTrack: Efficient and Precise Dynamic Race Detection", *PLDI'09*, pp. 121–133, 2009.

[10] D. Marino, M. Musuvathi, S. Narayanasamy, "LiteRace: Effective Sampling for Lightweight Data-Race Detection", *PLDI'09*, pp. 134–143, 2009.

[11] Y. Cai, W.K. Chan, "LOFT: Redundant Synchronization Event Removal for Data Race Detection", *22nd International Symposium on Software Reliability Engineering, IEEE*, pp. 160–169, 2011.

[12] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, "Eraser: A Dynamic Data Race Detector for Multithreaded Programs", *ACM TOCS*, vol. 15, no. 4, pp. 391–411, 1997.

[13] T. Elmas, S. Qadeer, S. Tasiran, "Goldilocks: a Race and Transaction-Aware Java Runtime", *PLDI'07*, pp. 245–255, 2007.

[14] P. Andrianov, V. Mutilin, and A. Khoroshilov, "An approach to lightweight static data race detection", *Proceedings of the Spring/Summer Young Researchers' Colloquium on Software Engineering, ISP RAS*, vol. 8, 2014.

[15] X.W. Xie, J.L. Xue, "ACCULOCK: Accurate and Efficient Detection of Data Races", *CGO'11*, pp. 201–212, 2011.

[16] Y. Yu, T. Rodeheffer, W. Chen, "RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking", *SOSP'05*, pp. 221–234, 2005.

[17] A. Jannesari, K. Bao, V. Pankratius, W. F. Tichy, "Helgrind+: An Efficient Dynamic Race Detector", *IPDPS'09*, pp. 1–13, 2009.

[18] (2023) The Sanitizers project. [Online]. Available: https://github.com/google/sanitizers

[19] K. Serebryany, T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice", *Proceedings of the Workshop on binary instrumentation and applications*, pp. 62-71, 2009.

[20] (2023) The Valgrind Project. [Online]. Available: https://valgrind.org

[21] (2023) Helgrind (The Valgrind Manual). [Online]. Available: https://valgrind.org/docs/manual/hg-manual.html

[22] (2023) DRD (The Valgrind Manual). [Online]. Available: https://valgrind.org/docs/manual/drd-manual.html

[23] J. Devietti et al., "Explicitly parallel programming with shared-memory is insane: at least make it deterministic!", *Proceedings of SHCMP*, 2008.

[24] B. Kasikci, C. Zamfir, G. Candea. "Data Races vs. Data Race Bugs: telling the difference with Portend", *ACM SIGPLAN Notices*, vol. 47, no. 4, pp. 185–198, 2012.

[25] J. Huang, C. Zhang, J. Dolby, "CLAP: Recording Local Executions to Reproduce Concurrency Failures", *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 141–152, 2013.

[26] N. Machado, B. Lucia, L. Rodrigues, "Concurrency Debugging with Differential Schedule Projections", *ACM SIGPLAN Notices*, vol. 50, no. 6, pp. 586–595, 2015.

[27] N. Machado, B. Lucia, L. Rodrigues, "Production-guided Concurrency Debugging", *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–12, 2016.

[28] C. Zamfir, G. Candea, "Execution Synthesis: a Technique for Automated Software Debugging", *Proceedings of the 5th European conference on Computer systems*, pp. 321–334, 2010.

[29] G. Altekar, I. Stoica, “ODR: Output-deterministic replay for multicore debugging”, *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 193–206, 2009.
[30] H. Cui et al., “Stable Deterministic Multithreading through Schedule Memoization”, *OSDI*, vol. 10, pp. 1–13, 2010.