The Quest for Precision: A Layered Approach for Data Race Detection in Static Analysis

Jakob Mund[1]*, Ralf Huuck[2], Ansgar Fehnker[2,3], and Cyrille Artho[4]

[1] Technische Universität München, Munich, Germany
mund@in.tum.de
[2] NICTA, University of New South Wales, Sydney, Australia
ralf.huuck@nicta.com.au
[3] University of the South Pacific, Suva, Fiji
ansgar.fehnker@usp.ac.fj
[4] Research Institute for Secure Systems, AIST, Amagasaki, Japan
c.artho@aist.go.jp

Abstract. Low level data-races in multi-threaded software are hard to detect, especially when requiring exhaustiveness, speed and precision. In this work, we combine ideas from run-time verification, static analysis and model checking to balance the above requirements. In particular, we adopt a well-known dynamic race detection algorithm based on calculating lock sets to static program analysis for achieving exhaustiveness. The resulting data race candidates are in a further step investigated by model checking with respect to a formal threading model to achieve precision. Moreover, we demonstrate the effectiveness of the combined approach by a case study on the open-source TFTP server OpenTFTP, which shows the trade-off between speed and precision in our two-stage analysis.

Keywords: Software verification, static analysis, concurrency, lock sets

1 Introduction

Modern processors commonly feature multi-core architectures. To fully use such hardware, software for multi-core processors often manages threads in the application code. Such concurrency carries the risk of introducing subtle but serious defects that might show up only sporadically and are extremely hard to debug. Common programming languages such as C provide only basic primitives for concurrency in terms of threading, while at the same time offering only limited tool support for debugging and bug prevention.

In this work we present a new way of detecting data races in embedded source code. We combine ideas from run-time verification, static analysis and software model checking by balancing their strengths and weaknesses. Run-time verification provides a good means to detect certain race conditions, but can only reason over program executions that have actually been observed, limiting

* This work was carried out while visiting NICTA.

```
1 int shared_var = 0;
2
3 void *reader() {
4   for(;;) {
5     t = shared_var;
6 } }
7
8 void *writer() {
9   for(;;) {
10    shared_var = compute();
11 } }
```

Fig. 1. Data Race Example

coverage. Static analysis is strong in covering all potential execution paths, but is prone to (over-)approximations leading to false positives. On the other hand, software model checking offers a precise analysis of the program semantics, but with limitations regarding scalability to larger code bases.

We propose a layered approach: In a first step we use a path-sensitive static implementation of the well-known *Eraser* algorithm [15]. Our static version is able to detect data races in C programs with a complete path coverage. While the algorithm is applicable to large-scale software, it is also prone to false positives. Therefore, in a second step we take those data races as *candidates* for a deeper model checking approach. The model checking phase abstracts from non-essential data and instructions and takes the threading model into account.

In this way we avoid to apply traditional software model checking to the full multi-threaded source code, but rather treat its application as a false-positive elimination step on selected code parts only. As a result we obtain a solution that can deal with real software systems, has a higher degree of coverage than run-time verification, but is more precise than traditional static analysis.

This paper is organized as follows: In Section 2 introduces to data races and the objectives of this work, together with related work. We present our two-step analysis approach in Section 3, covering the Eraser lock set analysis and its combination with software model checking. Experimental evaluation is presented in Section 4, followed by our conclusions in Section 5.

2 Data Races in Multi-threaded Programs

Threads are concurrent streams of program execution that can be created, merged and deleted at run-time. Threads might have access to shared resources. A *data race* occurs if two or more threads can simultaneously and non-atomically access a shared resource with at least one access being a write operation.

An example data race is given in Figure 1. A `reader` thread reads a shared variable (lines 3–6); a `writer` thread writes to it (lines 8–11). If these accesses are not synchronized using locks or other coordination mechanisms, then their effects are not well-defined. The update of the writer thread may become visible to other threads immediately or at any time after it has been issued, due to memory caches and other optimizations in modern hardware. Reading `shared_var` may thus yield different results depending on thread scheduling and hardware, which is why it is desirable to avoid data races in concurrent software.

As threads can be created dynamically, the number of threads may be large. The effect of data races may only be visible under a particular interleaving of thread actions at run-time; this makes the detection of data races difficult.

2.1 Scope and Contribution

Like the Eraser algorithm [15] used in run-time verification, we detect low-level data races by finding inconsistent or absent locking of variables shared across threads. Eraser monitors the *lock set* that protects a shared variable during read and write accesses to it. On every access, Eraser computes the intersection of the lock set protecting that variable. If the intersection becomes empty, i. e., there is no single lock consistently protecting a variable, a warning is issued.

Static Eraser Implementation. In this work we introduce a path-sensitive static implementation of Eraser. Unlike run-time verification we consider all paths statically, possibly over-approximating the set of feasible interleavings, but ensuring full coverage. This approach finds all data races but may issue spurious warnings.

Model-checking Thread-Interleaving. We also propose another analysis that is more precise and can reduce false positives from the previous step. The second analysis creates the *thread-interleaving graph* of the program (with limited depth) that captures the call structure of the threads and their termination, as well as the read and write accesses to shared variables. Since the interleaving graph grows exponentially with the number of threads in the program, we restrict it to the *data-race candidates* as identified in the static Eraser approach. We then model-check the thread-interleaving graph for feasible data races. This approach is sound up for a bounded number of threads. Using abstraction we are able to apply this methods to real code as shown in the evaluation in Section 4.

Our approach is based on a thread-interleaving semantics as defined in [12]. This semantics takes into account thread creation, join and cancellation as well as the acquisition and releases of locks. Moreover, it includes the advanced concepts of waiting and signaling that require a view of the global program state and is thus not covered by Eraser or other thread-modular approaches.

2.2 Related Work

Eraser [15] is the classical lock-set based algorithm that approximates potential data races very well, while not having the overhead of more precise but heavy-weight approaches based on the happens-before relation [16].

Goldilocks is a newer algorithm that computes data races precisely [7]. To be more accurate than Eraser, Goldilocks requires more elaborate data structures. Furthermore, the precision of Goldilocks depends on its ability to recognize overlapping data of multiple software transactions. That data is readily available and precise when using run-time verification, but is difficult to approximate precisely enough in static analysis, which is why our analysis is based on Eraser.

Other concurrency errors may still exist even in the absence of data races (called low-level data races to compare them with similar concurrency problems). High-level data races [3] and atomicity violations [2, 9] are two types of such problems. High-level data races cover non-atomic accesses to sets of dependent variables (multiple memory locations). Atomicity violations concern the scope during which a lock is held, and thus the use of the data rather than its direct access. These two types of problems have recently been subsumed by the notion of *causality* in data flow, which can cover both accesses to data and its use [6].

Static analysis of such concurrency properties has been attempted in other work, in a static analyzer where the rules are hard-coded in the program [1], and in a framework that is specialized for concurrency properties [11]. In contrast to that tool, we build on top of a general static analysis framework, Goanna [8], that allows flexible rules to express a wide range of different properties.

The second part of our work is closely related to other software model checkers, e. g., Java PathFinder [18] for Java bytecode and Inspect for C source code [19]. The key difference is that these software model checkers execute the full software at run time and explore alternative interleavings by rolling back the system to a previously stored state. This dynamic analysis is much more expensive than our approach, which works on an abstract model of the program.

Software model checkers working on a higher level of abstraction exist as well, such as SLAM, which analyzes device drivers against a given environment model [4], or SATABS, which can analyze programs using a subset of the Pthreads library [5]. In comparison, our work is not limited to certain domains (such as device drivers) and covers the full Pthreads library.

3 A Layered Approach for Static Race Detection

Our layered approach to detect data races first applies static analysis to obtain data race candidates and then applies model checking on those candidates.

3.1 Static Data Race Analysis

A common way to prevent data races is to impose a *locking discipline* that requires any shared (write) variable to be protected by at least one distinct lock among *all* threads. Since each lock can only be held by one thread, data races are effectively prevented.

Eraser [15] monitors the dynamic execution paths of each tread and records for each variable the lock set being held. If the intersection of those lock sets across threads for the same variable is empty, we assume a potential data race.

To achieve the same statically, we propose to check *all program paths* for each thread and then build the same intersection over all threads. Obviously, the static approach is an over-approximation as not all paths might be executable. We use the definition of a *lock set* [15, 14] as the mapping of shared variables to its potential set of locks, i. e., $\mathbf{Lockset} : \mathbf{Variables}_\star \rightarrow \wp(\mathbf{Locks}_\star)$. In the following, we show how to compute and check for emptiness of the **Lockset**.

Algorithm 1: Static implementation of the lock set algorithm.

```
begin
    Lockset(v) ← Locks_⋆;
    isReadOnly_v ← tt ;
    foreach π_i ∈ Threads_⋆ do
        lockstate ⟵ MFP(Nodes_{π_i}, is_locked);
        foreach n ∈ Nodes_{π_i} do
            if n accesses v then
                LocalLockset_{π_i}(v) ← LocalLockset_{π_i}(v) ∩ lockstate(n) ;
                isReadOnly_v ← isReadOnly_v ∨ (v is modified in n);
        Lockset(v) ← Lockset(v) ∩ LocalLockset_{π_i}(v);
```

Path-Sensitive Lock Set Computation. A thread (procedure) π is defined as a procedure with name pn that occurs in a thread-creation action denoted $\texttt{create}(\theta, pn)$. Nodes in the control flow graph of π are denoted by $\mathbf{Nodes}_\pi$. For a given thread π we define a function $\mathbf{is_locked} : \mathbf{Nodes}_\pi \times \mathbf{Locks}_\star \to \mathbb{B}$ that returns for each node n in π and each lock l whether l is held along all paths leading to n by

$$\mathbf{is_locked}(n, l) = \begin{cases} tt & \text{if } n = \mathbf{Lock\ l}, \\ ff & \text{if } n = \mathbf{Unlock\ l}, \\ \forall m \in pred(n) \bigwedge \mathbf{is_locked}(m, l) & \text{otherwise.} \end{cases}$$

Here *pred* denotes the predecessors of a node; the conjunction ensures coverage of all potential paths. This notion captures a standard path-sensitive static program analysis to compute the must-hold locks for each node in a thread.[5] Based on the information about the held locks, the thread-local lock set for each shared variable $v \in \mathbf{Variables}_\star$ and thread $\pi_i \in \mathbf{Threads}_\star$ is computed by

$$\mathbf{LocalLockset}(v, \pi_i) = \begin{cases} \bigcap_{n \in N_v} \{l' \mid \mathbf{is_locked}(n, l')\} & \text{if } N_v \neq \emptyset \text{ in } \pi_i, \\ \mathbf{Locks}_\star & \text{otherwise.} \end{cases}$$

where N_v denotes the set of all nodes of π_i which access variable v. The second case accounts for variables which are not accessed in π_i, mapping them to the set of all locks. Finally, the lock set for a program is the intersection of the lock sets for each thread occurring in a given program, i. e.,

$$\mathbf{Lockset}(v) := \bigcap_{\pi_i \in \mathbf{Threads}_\star} \mathbf{LocalLockset}(v, \pi_i)$$

Our algorithm [12] calculates the lock set using the maximal-fix-point worklist algorithm MFP [10] (see Algorithm 1). If $\mathbf{Lockset}(v)$ is non-empty for any shared variable v, the program is free of data-races.

[5] Modern programming languages like Java support `synchronized` blocks that acquire (resp. release) a lock when entering (resp. exiting) the critical section, enabling path-insensitive approaches [13].

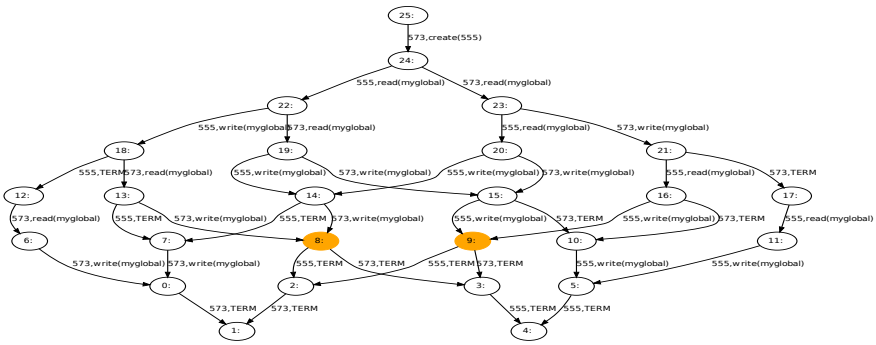

Fig. 2. A labeled transition system generated by applying the threading model. Colored vertices designate states with an imminent data race.

Soundness and Completeness Under the assumption that shared variables (as well as locks and signals) do not alias, the static lock set algorithm for low-level data race detection presented here is *sound*, hence false negatives (i. e., missed races) are not possible due to the locking discipline.

However, the analysis is *incomplete* as false positives (i. e., spurious warnings) are possible, because the analysis does not consider the *temporal* (also called *happens-before*) relation among events in different threads. Furthermore, warnings may be spurious if data races are avoided by other synchronization primitives like signals, or other more fine-grained accesses of variables [17].

3.2 Model-Checking of the Threading Semantics

While the approach presented in the previous section uses a fast thread-modular analysis, it is also prone to potential false-positives. An alternative precise approach is to use formal semantics for the multi-threading constructs and variable accesses and apply model-checking on the imposed model.

Semantics of Multi-Threaded Programs For model-checking we formalize the essence of multi-threaded program constructs [12], e. g., thread management and lock synchronization, according to structural operational semantics. This results in a labeled transition system, in which a data race happens if for a global state σ there are two threads θ_1, θ_2 and at least one of the threads is write-enabled on a shared variable v, while the other one can read or write to v:

$$\mathit{datarace}(\theta) = \mathit{enabled}(\texttt{write}_v, \theta_1, \sigma) \wedge (\mathit{enabled}(\texttt{write}_v, \theta_2, \sigma) \vee \mathit{enabled}(\texttt{read}_v, \theta_2, \sigma))$$

Using this predicate low-level data races can be detected by checking (on-the-fly) whether there is a path such that a data race can be reached.

An example of such a transition system is shown in Figure 2. Global configurations are numbered nodes; the transition system considers program actions relating to shared variable *myglobal*. Procedure *TERM* denotes successful termination; the number in a label represents the global transition relation (see [12]).

Algorithm 2: On-the-fly reachability checking using BFS.

```
begin
    Σ_worklist ← {⟨{θ_main ↦ Π(main)}, κ_∅, λl.⊥⟩};
    Σ_visited ← ∅;
    while Σ_worklist ≠ ∅ do
        σ_current ← dequeue(Σ_worklist);
        if σ_current ⊨ φ then
            WARN(σ_current);
            return true
        foreach σ' ∈ {σ' | σ_current --θ,a-->_G σ' ∧ σ' ∉ Σ_visited} do
            enqueue(σ', Σ_worklist);
        enqueue(σ_current, Σ_visited);
    return false
```

Global states 8 and 9 represent locations at which data races happen, because the two preceding actions in both states are unsynchronized write accesses.

On real systems, model construction potentially results in an exponential blow-up both in the numbers of threads and the number of thread operations. Moreover, thread creation in an (unbounded) loop may yield a possibly unbounded number of threads. In practice, model-checking would use a k-bound to restrict the number of threads that a single thread can create *per local state.*

Our interleaving semantics is a faithful abstraction of the real program by only considering thread-specific concepts and read/writes to shared variables. Mapping a threaded program to this abstraction is a non-trivial task when considering function calls and some subtleties of the POSIX standard.

Implementation Algorithm 2 checks whether a configuration satisfying a given predicate $\phi : \Sigma \to \mathbb{B}$ is reachable, and warns if a configuration satisfying ϕ ($\sigma \vDash \phi$) is found. In this algorithm, the reachable states of the model are not generated *a priori* but during the analysis itself, i. e., on the fly. This happens at the **foreach**-loop where solely the immediate successors of $\sigma_{current}$ are explored.

Unlike Algorithm 1, model checking yields precise diagnostic information. Line numbers shown by `WARN` substantially facilitate tracking down defects.

The use of a breadth-first-search was motivated by how thread interleaving influences the model. Different interleavings for the termination of threads constitute a large part of the model but are of less interest for race detection.

Soundness and Completeness Under the assumptions given above, our approach is *sound and complete* up to the fixed thread bound k, i. e., if each program instruction that instantiates a thread is successfully executed at most k times. Imprecision is introduced whenever thread instantiation is nested within loops that exceed the thread bound during execution. In those cases the analysis is neither sound nor complete. Fortunately, such bugs manifest rarely in practice.

Table 1. Evaluation results on OpenTFTP.

Analysis	# Races	Correct/Incorrect	T_{MTA}	T_{total}	$\frac{T_{\mathrm{MTA}}}{T_{\mathrm{total}}}$	$\frac{T_{\mathrm{MTA}}}{\#Vars}$	$\frac{T_{\mathrm{MTA}}}{kLOC}$
Lock set	15	4/11 (27 %)	7.58 s	38.63 s	19.6 %	0.47 s	3.03 s
Combined	0	n.a.	131.49 s	153.86 s	85.4 %	8.21 s	51.94 s
Combined*	4	4/0 (100 %)	2176.85 s	2194.36 s	99.2 %	136.05 s	869.69 s

3.3 Combining Both Analyses: The Layered Approach

The lock set algorithm is designed for performance, at the cost of possible spurious warnings. Since it is sound, each variable for which the analysis yields a non-empty set of distinct locks protecting it, is regarded as safe. On the other hand, model-checking yields precise results. However, the state-explosion problem often renders (detailed) models of concurrent programs too large for model-checking.

A natural consequence is to use a combined layered approach:

1. The lock set algorithm yields a (global) lock set for each shared variable. Variables with a non-empty lock set are safe.
2. We apply model-checking to the remaining shared variables in isolation. If a data race is reachable, we report that data race.

Step 2 can be thought of as a false-positive elimination for step 1. Note that the lock-set analysis does not worsen the precision of model-checking. It can be formally shown that if a non-empty lock set is found, a data race cannot be detected using the model-checking approach [12].

4 Case Study OpenTFTP

The core ideas of our layered approach have been implemented on top of the industrial-strength analysis tool *Goanna* [8]. Goanna analyzes C/C++ code using static analysis and model checking to detect bugs in large scale code. For our purposes we made use of the fact that the tool can readily produce control flow graphs, allows model generation with custom labels based on syntactic abstraction, and supports a summary-based interprocedural analysis.

However, a number of simplifications were made: The maximum thread creation bound was set to two, a pre-processing heuristics was used to detect the shared variables, and potential aliases as well as dynamic memory allocations were ignored. Moreover, for handling the threading semantics we inlined function calls, which is clearly not scalable, but sufficient for experiments.

The case study was executed on a Mobile Core2Duo Processor (clock freq. 1.83 Ghz, 4 GB of memory) running on Ubuntu Linux 9.10. We measured both the runtime of the multi-threading analyses presented in this paper (T_{MTA}) as well as the complete tool runtime including computation by Goanna (T_{total}).

The TFTP server software OpenTFTP was used as a real-world software example. The size of the program is about 2.5 KLOC, and it features high

functional complexity coupled with many multi-threading and synchronization-related constructs. Worker threads handle incoming requests, and shared resources like sockets are protected using mutexes. Furthermore, structured data types (`structs`) are used, whose impact on the precision can be evaluated. Obviously, analysis required an interprocedural setting to obtain meaningful results.

Out of 23 globally defined variables, 16 were identified as potentially shared and written to by at least on concurrent thread. Two distinct threads were identified, one being the main thread while the other is the `processRequest` worker-thread which is started for each incoming request; hence, thread creation is nested inside a loop. The initial run reported a multi-threaded control-flow graph with 2,339 distinct control-states and 3,766 transitions, and data races on 15 out of 16 shared variables.[6] Inspection revealed that the software was not programmed with respect to the POSIX standard, but with respect to some hidden assumptions on Linux, exploiting the fact that concurrently executing threads can release any lock held by any thread. We adjusted our model for this; the modified approach is denoted *Combined**, yielding precise results. Table 1 shows the results for the data race analysis using the lock set algorithm, the combined approach based on the POSIX standard, and the modified model based on the specific implementation on Linux exploited by the software.

5 Conclusion and Future Work

We propose a static implementation of the Eraser lock-set algorithm to detect possible data races in software. This analysis is sound but may result in false warnings. We add a second analysis step that model checks if potential data races detected by the lock-set analysis, can ever occur during program execution. Our two-step analysis takes into account the semantics of the Pthreads library, and is precise at the cost of a higher analysis overhead.

In future work, the performance of the second step could be improved further: As we consider only reachability properties, we could apply a strong partial-order reduction by abstracting from all concrete sequences of actions and considering only all possible global states. Moreover, instead of inlining procedures, some enriched summary information should be sufficient.

Other future work includes the analysis of other concurrency properties, such as deadlocks or high-level data races. Finally, the layered approach presented in this work may be applicable to other properties where fast over-approximations exist, making it be possible to balance speed and precision in a similar way.

Acknowledgments. NICTA is funded by the Australian Government (Department of Broadband, Communications and the Digital Economy) and the Australian Research Council through the ICT Centre of Excellence program.

[6] A multi-threaded control-flow graph embeds subgraphs of child threads into calls to `pthread_create`. The states and transitions thus correspond to local states; the number of global states is exponential in the number of local states and threads.

References

1. C. Artho and A. Biere. Applying static analysis to large-scale, multithreaded Java programs. In *Proc. 13th ASWEC*, pages 68–75, Canberra, Australia, 2001. IEEE.
2. C. Artho, A. Biere, and K. Havelund. Using block-local atomicity to detect stale-value concurrency errors. In *Proc. ATVA 2004*, volume 3299 of *LNCS*, pages 150–164, Taipei, Taiwan, 2004. Springer.
3. C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability (STVR)*, 13(4):220–227, 2003.
4. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. *SIGPLAN Not.*, 36(5):203–213, 2001.
5. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS 2005*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.
6. R. Dias, V. Pessanha, and J. Loureno. Precise detection of atomicity violations. In *Proc. HVC 2012*, LNCS. Springer, 2012.
7. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. *SIGPLAN Not.*, 42(6):245–255, 2007.
8. A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Model checking software at compile time. In *Proc. TASE 2007*. IEEE, 2007.
9. C. Flanagan, S. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, 2008.
10. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Progr. Lang. Syst.*, 18(3):299, 1996.
11. J. Loureno, D. Sousa, B. Teixeira, and R. Dias. Detecting concurrency anomalies in transactional memory programs. *Computer Science and Information Systems*, 8(2), 04 2011.
12. J. Mund. *Finding Common Defects in Multi-Threaded Programs at Compile Time*. PhD thesis, University of Augsburg, 2010.
13. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proc. PLDI 2006*, pages 308–319. ACM, 2006.
14. P. Pratikakis, J. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. *ACM SIGPLAN Notices*, 41(6):320–331, 2006.
15. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. on Computer Systems*, 15(4):391–411, 1997.
16. E. Schonberg. On-the-fly detection of access anomalies. In *In Proc. PLDI 1989*, pages 285–297, New York, NY, USA, 1989. ACM.
17. H. Seidl and V. Vojdani. Region analysis for race detection. *Static Analysis*, pages 171–187, 2010.
18. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
19. C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *Proc. ATVA 2008*, volume 5311 of *LNCS*, pages 126–140. Springer, 2008.