Dynamic information-flow analysis for multi-threaded applications

Laurent Mounier and Emmanuel Sifakis

VERIMAG laboratory – University of Grenoble
2 Av. Vignate, 38610 Gieres, France
{mounier,esifakis}@imag.fr

Abstract. Information-flow analysis is one of the promising techniques to leverage the detection of software vulnerabilities and confidentiality breaches. However, in the context of multi-threaded applications running on multicore platforms, this analysis becomes highly challenging due to data races and inter-processor dependences. In this paper we first review some of the existing information-flow analysis techniques and we discuss their limits in this particular context. Then, we propose a dedicated runtime predictive approach. It consists in extending information-flow properties computed from a single parallel execution trace to a set of valid serialisations with respect to the execution platform. This approach can be applied for instance in runtime monitoring or security testing of multi-threaded applications.

1 Introduction

On-going advances in processor technology and computer design allow to drastically reduce the cost of computing power and make it available to a large audience. As an example, multi-core architectures are now commonly used in many end user domains, ranging from small embedded devices like smart-phones to powerful personal computers. To correctly exploit the huge computing capabilities of these machines, applications are conceived as a set of asynchronous tasks (or *threads*), able to execute on distinct processors, and cooperating each others to provide the desired functionalities. An example of such a parallel programming model is based on *shared memory* to implement inter-thread communications and synchronisations.

However, exploiting efficiently and correctly this hardware-supplied parallelism is notoriously difficult. In fact, the primitives offered by many classical programming languages to control asynchronous parallel executions are still basic and error prone. As a consequence, it is necessary to develop suitable techniques and tools allowing to analyse this kind of applications.

An important class of analysis is based on the notion of *information flow*. Their purpose is to track how data processed by a program can transit inside the memory at execution time. Such analysis are useful for many validation purposes, and it is a central issue in computer security. In particular it allows to detect *information leakage* (from a confidentiality point of view), or to compute *taint*

propagation (to check how user inputs may influence vulnerable statements). Information inside applications can flow in various ways, some of which being obvious and some others being more tedious to identify. *Explicit flows*, corresponding to assignments between variables, are the most commonly analysed. On the other hand, implicit flows using *covert channels* such as control flow and timing delays are much harder to detect.

As many program analysis techniques, information-flow analysis is much more challenging when considering parallel executions. This difficulty comes from several sources, including for instance:

- the extra flows introduced by inter-thread communication channels;
- the conflicting accesses to shared resources or memory locations between concurrent threads (e.g., race conditions);
- the non-determinism introduced by the execution platform (hardware and operating system), which makes some program executions hard to reproduce;
- etc.

Various software analysis techniques have been proposed so far to address these problems. These techniques are either *static* (they do not require any program execution), like data-flow analysis or model-checking, or *dynamic*, like runtime monitoring or test execution. The main difficulty here is to extend the analysis techniques used for sequential programs while avoiding the so-called "interleaving problems". These problems are related to the exponential blow-up occurring when considering all possible serialisations of a parallel execution.

More recent proposals, like *predictive runtime-analysis*, are based on ad hoc combinations of static and dynamic approaches. These techniques consist in extending the results obtained at runtime when observing a given parallel execution to a set of valid serialisations, corresponding to execution sequences that have not been observed, but that could have occurred. Thus, this set can be seen as a *slice* of the target program, computed from a single execution. However, most of the existing predictive runtime-analysis techniques focus on the effects of *coarse-grain* parallelism, introduced by inter-thread scheduling. This scheduling influences the execution order of concurrent eligible threads. Its decisions depend on non-controllable events (e.g., I/O latency), and therefore other interleavings could occur and they are taken into account by the analysis.

Another source of conflict is produced by the "simultaneous" execution of multiple instructions by several processors. Here, the conflicting accesses are (implicitly) solved by the execution platform, and this behaviour escapes from the program level. A possible way to handle this *fine-grain* parallelism is to rely on dedicated hardware elements, allowing to monitor the current execution at a very low level. Thus, specific architectures for dynamic information-flow tracking have been proposed.

In this paper we review some representative works (section 2) illustrating the information-flow analysis techniques stated above and we identify their benefits and limitations. The focus is essentially put on dynamic analysis techniques, and therefore we present some of the monitoring techniques available (section 3) and their use in the context of multi-core executions. Then, we propose a predictive

approach (section 4) to address fine-grain parallelism effects, without requiring a specific architecture. Finally, we give some conclusions and perspectives (section 5).

2 Information-flow analysis of multi-threaded programs

The importance of information flow has captured the interest of researchers working in various domains of computer systems. Starting from hardware, where special architectures have been conceived ([1,2,3]) to operating systems ([4]) and up to the application layer which is dominated by *static* and *dynamic* approaches detailed hereafter.

2.1 Static analysis techniques

Static approaches usually reason on source code level. For instance, a possible approach to secure a program execution against information leakage is to promote *type-safe* languages, as proposed by [5,6], to guarantee *by construction* secure information flows. In some cases these languages include primitives for multi-threaded development ([7,8]).

Regarding general static analysis techniques, a work direction was to extend the data-flow analysis techniques used for sequential code while avoiding the "interleaving explosion problem" mentioned in the introduction. A first way to address this problem was to take into account restricted forms of parallelism, like in [9] (no parallel loops) or in [10] (*cobegin/coend* primitives). However, an important step was made in [11]. In this work the authors proposed to consider a sub-class of data-flow analysis problems, the so-called *bit-vector problems*. They define an efficient generalisation of (unidirectional) bit-vector analysis to static parallel programs which faithfully captures the effect of inter-thread dependencies without requiring to enumerate each possible interleavings. The key assumption is to consider bit-vector properties that are *generated* on an execution trace by a *single* transition of the control-flow graph (and not by a combination of transitions). This allows to reduce the effects of inter-thread dependencies at the instruction level, without taking into account whole execution paths occurring in other threads. Note that this category of bit-vector problems is large enough to encompass many interesting properties, including information-flow analysis. More recently, this solution has been extended to deal with dynamic synchronisation primitives [12].

Other existing solutions for information-flow analysis rely on the computation of so-called "Program Dependency Graphs" (PDGs) to express data dependencies. PDGs for concurrent programs were first proposed in [13], based on the computation of *may-happen-in-parallel* (MHP) relations to approximate the effects of concurrent access to shared variables. Precise computations of MHP relations are known to be expensive. However, this static approach has been used in several works dedicated to information flow analysis ([14,15]).

2.2 Dynamic analysis techniques

Dynamic approaches may look more appealing for analysing multi-threaded applications. However, they require some instrumentation facilities to track information flows at execution time. There are several frameworks available (such as [16,17,18]) that facilitate the implementation of dynamic monitoring tools. More details about these frameworks are presented in section 3.

An interesting class of dynamic analysis techniques is the so-called *predictive runtime-analysis* category. They consist in observing/monitoring a single parallel execution sequence σ (as for sequential programs), and then to *generalise* the results obtained to other execution sequences corresponding to possible interleavings of σ (i.e., that could have been observed if another valid schedule occurred). This gives a kind of *program slice* of reasonable size, that can be handled by various techniques like static analysis [19], or even test generation [20]. Depending on the approach chosen to generalise the observed trace (and to represent the resulting set of serialisations), the program slice obtained may over-approximate or under-approximate the concrete program behaviour. A short survey on such runtime prediction techniques is provided in [21], together with a precise trace generalisation model.

Dynamic analysis techniques are widely used in the context of multi-threaded applications for runtime error detection like deadlocks ([22,23]) and data races ([24,25]. Although detecting data races could be useful for information-flow analysis, it is not sufficient as such. Hence, more focused analyses are developed to deal with malware detection ([26,27]) and enforcement of security policies ([28,29]).

3 Building tools for dynamic analysis

Building dynamic analysis tools necessitates integrating some monitoring facilities to the analysed application. Monitoring features are added either at source code level or binary level, either statically or dynamically. Waddington et al. [30] present a survey on these techniques.

Instrumentation code is often added statically in applications as implicit logging instructions. It necessitates access to the source code and can be added accordingly by the developers (which is a tedious and error-prone procedure) or automatically. To automate this process source-to-source transformations can be applied, for instance using aspect-oriented programming. Apart from the source level, static instrumentation can also be applied directly at the binary level, e.g., using frameworks like Dyninst [17]. Hereafter we take a closer look to dynamic binary instrumentation (DBI) techniques.

3.1 Dynamic binary instrumentation

In general, DBI frameworks consist of a front-end and a back-end. The front-end is an API allowing to specify instrumentation code and the points at which it

should be introduced at runtime. The back-end introduces instrumentation at the specified positions and provides all necessary information to the front-end.

There are two main approaches for controlling the monitored application: *emulation* and *just-in-time* (JIT) instrumentation. The emulation approach consists in executing the application on a *virtual machine* while the JIT approach consists in linking the instrumentation framework dynamically with the monitored application and inject instrumentation code at runtime.

Valgrind [18] is a representative framework applying the emulation approach. The analysed program is first translated into an intermediate representation (IR). This IR is architecture independent, which makes it more comfortable to write generic tools. The modified IR is then translated into binary code for the execution platform. Translating code to and from the IR is time consuming. The penalty in execution time is approximately four to five times (with respect to an un-instrumented execution).

Pin [16] is a widely used framework which gains momentum in analysing multi-threaded programs running on multi-core platforms. Pin and the analysed application are loaded together. Pin is responsible of intercepting the applications instructions and analysing or modifying them as described by the instrumentation code written in so-called pintools. Integration of Pin is almost transparent to the executed application.

The pintools use the frameworks front-end to control the application. Instrumentation can be easily added at various granularity levels from function call level down to processor instructions. An interface exists for accessing abstract instructions common to all architectures. If needed more architecture specific analyses can be implemented using specific APIs. In this case the analysis written is limited to executables of that specific architecture.

Adapting a DBI framework to parallel architectures is not straight forward. Hazelwood et al. [31] point out the difficulties in implementing a framework that scales well in a parallel environment and present how they overcame them in the implementation of Pin. As mentioned in their article, extra care is taken to allow frequently accessed code or data to be updated by one thread without blocking the others. Despite all this effort in some cases the instrumenter will inevitably serialise the threads execution or preempt them.

Another challenging issue is writing parallel analysis. The monitored data must also be updated in parallel, and data races on the monitored data should be eliminated.

3.2 Hardware-based monitoring techniques

The software instrumentation techniques described in the previous section suffer from practical limitations in a multi-thread context. In particular:

- they may introduce a rather huge time overhead (making the execution between 10 and 100 times slower [32,18,33]);
- they do not take into account the specific features of a multi-core execution;

– they do not exploit as much as possible all the computational resources of the execution platform.

To overcome these limitations, especially in the context of instruction-grain monitoring, several proposals have been made to introduce some dedicated hardware mechanisms. We discuss here some of these proposals.

First of all, let us recall that instruction grain monitoring is based on several steps:

– capturing the relevant events after each executed instruction;
– propagating these events to the monitor process (event streaming);
– updating the meta-data (or shadow memory);
– executing the appropriate checks.

Each of these steps is a potential source of overhead, and techniques have been proposed to optimise them at the hardware level.

Some of these acceleration techniques are not specific to multi-core executions. For instance Venkataramani et al. [34] proposes to add some extra pipeline stage to perform metadata updates and checks, [35,36] improves the management of metadata through micro-architectural changes. Another option is to reduce the binary instrumentation cost by means of special registers [37], or using cache line tags to trigger event handlers [38].

Regarding multi-core platforms, one of the main proposals is to take advantage of the processors availability to dedicate one (or several) cores to the monitoring task. This idea has been implemented for two typical instruction-level monitoring problems.

Shetty et al. [39] propose in their work a monitoring technique dedicated to memory bugs (e.g., memory leaks, unallocated memory errors, etc.). Its principle is to associate a *monitoring thread* to each *application thread*. On a multi-core platform, both threads can run in parallel on distinct cores. To improve the 2-way communication between these threads, dedicated FIFO buffers are used (instead of using shared memory): one buffer for check requests, and one buffer for check reports. When one of these buffers is full/empty the application/monitoring thread is stalled. Moreover, since the duty cycle of the monitoring thread may be low[1], it can be suspended at any time. Other optimisations include the use of a separate L2 cache for the monitoring thread in order to reduce cache contention. Evaluation results show a monitoring overhead less than 50%, depending on the considered architecture.

The work of Nagarajan et al. [40] aims to *enforce* taint-based security policies. The idea is to use a dedicated thread as a "shadow execution" to keep the taint value of each register and memory locations of the application thread. This monitoring thread interrupts the application thread when the taint policy is violated. Here again, the main difficulties are to keep synchronised both threads and to ensure communication between them. As in [39], a FIFO buffer is used. A specific problem is to correctly react in case of policy violation to ensure fail-safety (i.e., to stop the execution as soon as possible). The solution proposed uses

[1] it may not be the case when monitoring other properties

a 2-way communication between the two threads before each *critical* operation (with respect to the policy considered).

Finally, a more recent work [41] advocates the use of so-called *log-based architecture* as a suitable trade-off between *efficiency* (how reduced is the monitoring overhead) and *genericity* (how general is the monitoring support). In this proposal, each core is considered as a log producer/consumer during the execution. When an instruction is executed on producer core, a (compressed) *record* is computed to store relevant information (program counter, instruction type, operand identifiers and/or addresses). Each record may correspond to one or more events on the consumer side. Record transmission is achieved using a large (up to 1MB) log buffer. As a consequence, an implicit synchronisation occurs between producer and consumer threads when the buffer is full or empty (in the former case the application is stalled). This may introduce a (bounded) lag between the time a bug occurs, and the time it is detected. To improve metadata (e.g., taint values) tracking, another feature is to associate a (small) shadow register to each data register in order to store the addresses from which it inherits (rather than the data itself). This choice makes the tracking more general (suitable for more applications) while keeping it efficient. In addition, to reduce the numbers of checks, a dedicated event cache is used (when an event hits the cache it is considered as redundant and discarded). Finally, a rather sophisticated metadata memory layout is provided, with a new instruction allowing to directly translate a data address to its metadata counterpart.

Experiments performed in [42] with this architecture on several monitoring applications (taint analysis, data race detection, memory checking) show an overhead smaller than 50% can be obtained for CPU-intensive applications.

4 Extended Information-Flow Analysis

We present in this section an alternative approach to perform dynamic analysis on a multi-core execution. This approach fits in the category of (over-approximative) predictive runtime-analysis. Its purpose is to extend the results obtained from the observation of a (parallel) execution sequence $\sigma_{\parallel}$ to the set of all serialised execution sequences corresponding to valid interleavings of $\sigma_{\parallel}$. The interleaving we consider here are essentially the ones produced by "side effects" introduced by the execution platform. The goal is to extend an observed execution $\sigma_{\parallel}$ such that the effects of the hardware it was executed on are captured.

In fact, when executing an application in parallel on several cores, platform-related effects may "obfuscate" the observed execution trace $\sigma_{\parallel}$. This may happen for instance due to a cache miss which could delay the effect of an observed instruction to a shared memory location. Similarly, small local overheads introduced by the monitoring probes or by I/O operations may slightly perturb the execution schedule (i.e., the sets of concurrently executed instructions), changing the sequence of (shared) memory updates. As a result, one can legitimately consider that the observation of $\sigma_{\parallel}$ does not fully nor accurately represent a real (non monitored) parallel execution.

A possible way to take into account this uncertainty in the observed execution sequence, is to assume the existence of a bounded time interval δ, during which the effects of instructions executed by concurrent threads may interleave. This value δ depends on the execution platform we consider. We present hereafter the method we propose for taking into account all these possible serialisations of $\sigma_{\|}$ during a dynamic information-flow analysis.

4.1 The "butterfly" approach

The method we propose is partially inspired by the work proposed in [42]. We summarise here what are the main similarities and differences between these two approaches.

The main objective of [42] is to provide a *lifeguard* mechanism for (multi-threaded) applications running on multi-core architectures. It is a runtime enforcement technique, which consists in monitoring a running application to raise an alarm (or interrupt the execution) when an error occurs (e.g., writing to an unallocated memory). The main difficulty is to make the lifeguard reasoning about the *set* of parallel executions. To solve this issue, the authors considered (monitored) executions produced on specific machine architectures [41] on which *heartbeats* can be sent regularly as *synchronisation barriers*, to each core. This execution model can be captured by a notion of *uncertainty epochs*, corresponding to code fragments such that a *strict happens-before* execution relation holds between non-adjacent epochs. These assumptions allow to define a conservative data-flow analysis, based on sliding window principle, taking into account a superset of the interleaving that could occur in three consecutive epochs. The result of this analysis is then used to feed the lifeguard monitor. This approach can be used to check various properties like use-after-free errors or unexpected tainted variable propagation.

Our objectives are not the same. Our intention is to provide some *verdict* to be used in a property oriented test-based validation technique for multi-core architectures. As such, our solution does not need to be necessarily conservative: false negatives are not a critical issue. A consequence is that we do not require any specific architecture (nor heartbeat mechanism) at execution time. Another main distinction is that we may proceed in a *post-mortem* approach: we first produce log files which record information produced at runtime, then this information is analysed to provide various test verdicts (depending on the property under test). This makes the analysis more flexible by decoupling the execution part and the property checking part. From a more technical point of view, we also introduced some differences in the data-flow analysis itself. In particular we considered a sliding window of two epochs (instead of three). From our point view, this makes the algorithms simpler, without sacrificing efficiency. Finally, a further contribution is that we take into account the information provided by mutex locks to reduce the number of false positives.

4.2 A window-based information flow analysis

We present here the basis of our window-based dynamic information-flow analysis. More details can be found in [43]. Its goal is to extend the analysis verdict of σ_0 (the observed serialisation) to a set of valid serialisations σ_δ, where δ is a platform-dependent time interval representing the (maximal) overlap between instruction sequences executed in parallel. The main concern is to avoid the whole enumeration of this set. To that end, we use a *sliding window-based* approach. Each window contains a set of concurrent instruction sequences belonging to the active threads (the ones currently executing on a given core). The analysis technique consists in summarising the parallel execution up to the current window $\mathcal{W}$, and to update this summary by taking into account the effects of possible serialisations of the execution sequences belonging to $\mathcal{W}$. This update is performed by means of iterative fix-point algorithms, as explained below.

To properly define each window, we time slice σ_0 using arbitrary time intervals greater than δ. We call these time slices *epochs*. However, instructions at the boundaries of adjacent epochs (hence within a time distance smaller than δ) may interleave, according to our hypothesis. To take this into account we define windows of size two epochs, and we extend the interleaving assumption such that all instructions of a window may interleave. This extension ensures that our analysis results will actually capture the serialisations of a set σ_s, where $\sigma_s \supseteq \sigma_\delta$.

Fig 1 illustrates the parallel execution of two threads (A and B). The dots represent instructions. Each instruction can be identified by a triplet (l, t, j) where l is the epoch it was executed in, t is the thread that executed it and j is an identifier of the instruction inside t. Instructions executed by the same thread in an epoch are surrounded by a box which is a basic block identified as (l, t). The arrows originating from the (highlighted) instruction (l_b, B, i) illustrates our interleaving assumptions. We can note at the boundary between epochs l_h and l_b the definition of the time interval δ. The solid arrows capture the serialisations of (l_b, B, i) for all σ_δ and the dashed arrows the extended serialisations of (l_b, B, i) in σ_s.

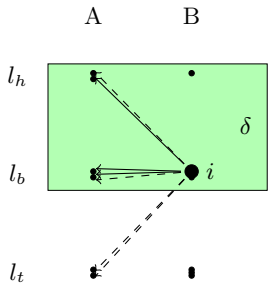

Fig. 1. Interleaving assumptions

4.3 Iterative information flow computation

As explained above, processing a window of two epochs means computing the effects produced by all possible serialisations of the parallel instruction sequences it contains (since all these serialisations are considered as valid). To do so, we use an iterative fix-point computation algorithm. This algorithm proceeds as follows:

- First, we define a sequential data-flow analysis of the property under check. This property could be for instance a taint-analysis, a memory consistency checking, a null-pointer analysis, etc. An important requirement is that this data-flow analysis should be expressed as a bit-vector problem (which is in fact the case for most of the analysis used in practice). Running this sequential data-flow analysis on a single thread t allows to update a given initial summary S_0 (expressed as a *state vector*) into a a new summary S_1^t.
- Since threads are not independent (they may share memory locations), the sequential analysis ran on each thread should be combined with the others. In other words, results produced by executing instruction (l, t, i) should be made available to all instructions (l, t', j) of the window for $t \neq t'$ (according to our assumptions). This could be achieved by *running again* each sequential analysis on each thread t, starting now from an initial state $S_1 = \cup_{t \neq t'} S_1^t$.
- This step is repeated as long as the summary is changed. Since we consider bit-vector problems, this process will eventually reach a fix-point.

This algorithm can be implemented using two generic procedures: a first one (vertical step) iterates the sequential analysis over each thread, the second one (horizontal step) runs the vertical step along the two epochs of the window. Since adjacent epochs may also interleave, the vertical step should be repeated until a (window-level) fix-point is reached. Depending on the analysis under consideration, further processing may be required to "clean up" the results produced (removing the effects of some non valid execution sequences).

It has been showed in [43] that for a taint-analysis:

- this algorithm detects all tainted variables;
- the set of variables detected can be split into *strongly tainted* variables (corresponding to variables really tainted), and *weakly tainted* variables (potential false positives). These false positives are due to our sliding window techniques which may over-approximate the set of valid serialisations across several windows.

4.4 Experimental results

The window-based methodology we presented can be applied both at runtime or off-line as a post-mortem analysis. We have implemented a tool chain for taint analysis using a post-mortem approach. The tool necessitates the source code of the multi-threaded application (written in C using *pthreads* library). Instrumentation code is added as logging instructions via a source to source

transformation. At execution time log files are generated containing address information on assignments.

For taint analysis the summary actually contains variables that can be tainted through a valid serialisation up to the preceding window. The window analysis must hence infer local serialisations (of instructions in window) which either taint new variables or untaint some existing. The serialisations are discovered through the iterative algorithm. Some special care must be taken though on how *gen*/*kill* information is propagated and how the summary of a window is computed.

Experimental results on small handcrafted benchmarks using five threads racing for access to a shared data structure show an overhead of 50% for producing the log files. The taint-analysis then takes less than 1 second to analyse about 5000 log lines on a Intel i3 CPU @2.4GHz with 3GB of RAM.

5 Conclusion

In this work we have discussed some issues regarding dynamic information-flow monitoring of multi-thread applications running on multi-core architectures. We gave a brief overview of the main existing techniques and underlying tools considered so far to address this issue. The general concerns are to limit the monitoring overhead at runtime, to avoid the explicit exploration of all possible execution sequence interleavings, and to propose general enough frameworks (able to handle various kinds of analysis). In our opinion, two directions are rather promising:

- runtime-prediction techniques, which allow to extend the results produced by a single (parallel) execution to a whole program slice consisting of valid "neighbour" executions;
- hardware-level optimisations of the monitoring techniques.

The former solution can be used in a general context, for instance in a test-based approach where the goal is to evaluate the "robustness" of the application on various execution conditions. The later solution is better suited for specific applications (e.g., with strong security or reliability requirements), and it provides an integrated hardware/software monitoring and enforcement framework.

We also proposed a prospective runtime-prediction technique. Its purpose is to deal explicitly with fine-grain interleavings produced by the multi-core execution platform. Experimental results obtained so far for taint-analysis are encouraging in terms of performance. Further work is now required to extend the prototype and consider other kinds of analysis.

References

1. Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., Rosenblum, M.: Understanding data lifetime via whole system simulation. In: Proceedings of the 13th conference on USENIX Security Symposium - Volume 13. SSYM'04, Berkeley, CA, USA, USENIX Association (2004) 22–22

2. Crandall, J.R., Wu, S.F., Chong, F.T.: Minos: Architectural support for protecting control data. ACM Trans. Archit. Code Optim. **3**(4) (December 2006) 359–389
3. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. SIGARCH Comput. Archit. News **32**(5) (October 2004) 85–96
4. Clemente, P., Rouzaud-Cornabas, J., Toinard, C.: Transactions on computational science xi. Springer-Verlag, Berlin, Heidelberg (2010) 131–161
5. Volpano, D., Smith, G.: A type-based approach to pro-gram security. In: In Proceedings of the 7th International Joint Conference on the Theory and Practice of Software Devel-opment, Springer (1997) 607–621
6. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications **21** (2003)
7. Barthe, G., Rezk, T., Russo, A., Sabelfeld, A.: Security of multithreaded programs by compilation. ACM Trans. Inf. Syst. Secur. **13**(3) (July 2010) 21:1–21:32
8. Smith, G., Volpano, D.: Secure information flow in a multi-threaded imperative language. In: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '98, New York, NY, USA, ACM (1998) 355–364
9. Grunwald, D., Srinivasan, H.: Data flow equations for explicitly parallel programs. In: PPOPP, ACM (1993)
10. Krinke, J.: Static slicing of threaded programs. SIGPLAN (1998)
11. Knoop, J., Bernhard, S., Vollmer, J.: Parallelism for free: efficient and optimal bitvector analyses for parallel programs. ACM Trans. Program. Lang. Syst. (1996)
12. Farzan, A., Kincaid, Z.: Compositional bitvector analysis for concurrent programs with nested locks. In: SAS, Springer-Verlag (2010)
13. Krinke, J.: Context-sensitive slicing of concurrent programs. SIGSOFT (2003)
14. H., C.: Information flow control for java based on path conditions in dependence graphs. In: Secure Software Engineering, IEEE Computer Society (2006)
15. Liu, Y., Milanova, A.: Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows. In: Software Maintenance and Reengineering, IEEE Computer Society (2010)
16. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. PLDI '05, New York, NY, USA, ACM (2005) 190–200
17. Buck, B., Hollingsworth, J.K.: An api for runtime code patching. The International Journal of High Performance Computing Applications **14** (2000) 317–329
18. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. In: Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), San Diego, California, USA (June 2007) 89–100
19. Ganai, M.K., Wang, C.: Interval analysis for concurrent trace programs using transaction sequence graphs. RV (2010) 253–269
20. Kundu, S., Ganai, M.K., Wang, C.: Contessa: Concurrency testing augmented with symbolic analysis. In: CAV, Springer (2010) 127–131
21. Wang, C., Ganai, M.K.: Predicting concurrency failures in the generalized execution traces of x86 executables. In: RV. (2011)
22. Li, T., Ellis, C.S., Lebeck, A.R., Sorin, D.J.: Pulse: a dynamic deadlock detection mechanism using speculative execution. In: Proceedings of the annual conference on

USENIX Annual Technical Conference. ATEC '05, Berkeley, CA, USA, USENIX Association (2005) 3–3
23. Castillo, M., Farina, F., Cordoba, A.: A dynamic deadlock detection/resolution algorithm with linear message complexity. In: Proceedings of the 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing. PDP '12, Washington, DC, USA, IEEE Computer Society (2012) 175–179
24. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4) (November 1997) 391–411
25. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. WBIA '09, New York, NY, USA, ACM (2009) 62–71
26. Bayer, U., Kirda, E., Kruegel, C.: Improving the efficiency of dynamic malware analysis. In: Proceedings of the 2010 ACM Symposium on Applied Computing. SAC '10, New York, NY, USA, ACM (2010) 1871–1878
27. Egele, M., Scholte, T., Kirda, E., Kruegel, C.: A survey on automated dynamic malware-analysis techniques and tools. ACM Comput. Surv. **44**(2) (March 2008) 6:1–6:42
28. Zhu, D.Y., Jung, J., Song, D., Kohno, T., Wetherall, D.: Tainteraser: protecting sensitive data leaks using application-level taint tracking. SIGOPS Oper. Syst. Rev. **45**(1) (February 2011) 142–154
29. Cristia, M., Mata, P.: Runtime enforcement of noninterference by duplicating processes and their memories. In: WSEGI. (2009)
30. Waddington, Roy, Schmidt: Dynamic analysis and profiling of multi-threaded systems
31. Hazelwood, K., Lueck, G., Cohn, R.: Scalable support for multithreaded applications on dynamic binary instrumentation systems. In: Proceedings of the 2009 international symposium on Memory management. ISMM '09, New York, NY, USA, ACM (2009) 20–29
32. Nethercote, N.: Dynamic Binary Analysis and Instrumentation. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom (November 2004)
33. Uh, G.R., Cohn, R., Yadavalli, B., Peri, R., Ayyagari, R.: Analyzing dynamic binary instrumentation overhead. In: Workshop on Binary Instrumentation and Application, San Jose, CA (October 2007)
34. Venkataramani, G., Roemer, B., Solihin, Y., Prvulovic, M.: Memtracker: Efficient and programmable support for memory access monitoring and debugging. In: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture. HPCA '07, Washington, DC, USA, IEEE Computer Society (2007) 273–284
35. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. SIGPLAN Not. **39**(11) (October 2004) 85–96
36. Venkataramani, G., Doudalis, I., Solihin, Y., Prvulovic, M.: Flexitaint: A programmable accelerator for dynamic taint propagation. In: 14th International Symposium on HighPerformance Computer Architecture. (2008)
37. Corliss, M.L., Lewis, E.C., Roth, A.: Dise: a programmable macro engine for customizing applications. SIGARCH Comput. Archit. News **31**(2) (May 2003) 362–373
38. Zhou, Y., Zhou, P., Qin, F., Liu, W., Torrellas, J.: Efficient and flexible architectural support for dynamic monitoring. ACM Trans. Archit. Code Optim. **2**(1) (March 2005) 3–33

39. Shetty, R., Kharbutli, M., Solihin, Y., Prvulovic, M.: Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. IBM J. Res. Dev. **50**(2/3) (March 2006) 261–275
40. Nagarajan, V., H-S.Kim, Y.Wu, Gupta, R.: Dynamic information flow tracking on multicores. In: Workshop on Interaction between Compilers and Computer Architectures, Salt Lake City (February 2008)
41. Chen, S., Kozuch, M., Strigkos, T., Falsafi, B., Gibbons, P.B., Mowry, T.C., Ramachandran, V., Ruwase, O., Ryan, M., Vlachos, E.: Flexible hardware acceleration for instruction-grain program monitoring. In: Proceedings of the 35th Annual International Symposium on Computer Architecture. ISCA '08, Washington, DC, USA, IEEE Computer Society (2008) 377–388
42. Goodstein, M.L., Vlachos, E., Chen, S., Gibbons, P.B., Kozuch, M.A., Mowry, T.C.: Butterfly analysis: adapting dataflow analysis to dynamic parallel monitoring. In: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems. ASPLOS '10, New York, NY, USA, ACM (2010) 257–270
43. Sifakis, E., Mounier, L.: Extended dynamic taint analysis of multi-threaded applications. Technical report, VERIMAG, University of Grenoble (June 2012)