Offline taint prediction for multi-threaded applications.

Emmanuel Sifakis, Laurent Mounier

Verimag Research Report n^o TR-2012-08

March 20, 2013

Unité Mixte de Recherche 5104 CNRS - Grenoble INP - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Offline taint prediction for multi-threaded applications.

Emmanuel Sifakis, Laurent Mounier

March 20, 2013

Abstract

Dynamic analysis of multi-threaded applications running on parallel architectures is notoriously a challenging issue. In this work we consider taint analysis as a typical information flow property. The approach we propose extends properties collected at runtime on a *single* parallel execution $\sigma_{\|}$ to *a set of* execution sequences corresponding to plausible serializations of $\sigma_{\|}$. Taint values are inferred using a sliding-window based static analysis, performed on a fragment of an execution trace. We provide sufficient conditions to reduce some of the false positives produced by the over-approximation of serializations. Only explicit taint propagation is captured but special care has been taken to handle lock-based critical sections correctly. A proof-of-concept implementation has been developed using the CETUS framework, and some experimental results are given. Finally, the framework could be extended to perform other types of information flow analysis.

Keywords: multithread,dynamic/static analysis,mutex,lock,taint

Reviewers:

How to cite this report:

@techreport {TR-2012-08,
title = {Offline taint prediction for multi-threaded applications.},
author = {Emmanuel Sifakis, Laurent Mounier},
institution = {{Verimag} Research Report},
number = {TR-2012-08},
year = {2012}
}

1 Introduction

Taint analysis is a very popular dynamic information flow analysis. It consists into marking/tainting and following at runtime their propagation inside a program. Taint analysis is widely used in several contexts such as vulnerability detection, information policy enforcement, testing and debugging. A significant effort has been put on optimizing taint analysis for sequential programs. However, adapting taint analyses to programs executing in parallel is utterly difficult due to non-deterministic memory accesses.

1.1 Exploiting parallelism

Parallelism is often exposed to programmers through the notion of *threads*, or light weight processes, which are schedulable streams of code. The thread programming model is well established and widely used even on mono-processor architectures. It provides a clear separation of tasks, while allowing their efficient interaction and synchronization through *shared memory*. The execution of multithreaded programs on multicore platforms is capable of exploiting the underlaying parallelism. Thus, multithreaded applications running on multicore architectures are in common use, and there is an increasing need for suitable validation tools. However, validating parallel code not only inherits all difficulties of sequential code validation, but also amplifies them due to inter-thread dependencies caused by shared data (data races).

Parallel execution of threads introduces new issues to software validation. In the recent past, threads were executed sequentially interleaved by the *operating system* thus, giving the illusion of a parallel execution. This execution model is simpler to validate because a single thread is executed at any time. Although data races may occur due to non-deterministic scheduling options, at least the effects of memory accesses are easily observable. This is not the case in parallel execution of threads where each thread can observe a different serialization of events. The non-determinism of memory accesses that occur in parallel is platform dependent and it is described in the platforms memory consistency model.

We provide in Figure 1 an overview of multithreaded programs and their sequential and parallel execution. Initially, we define a multithreaded program $\mathfrak{P}$as a set of threads T^a, where each thread specifies a sequence of events to be executed. At the top of the figure we explicit an excerpt of a program $\mathfrak{P} = \{T^a, T^b, T^c\}$. Next, we give two plausible schedules for the excerpt of program $\mathfrak{P}$. A schedule is a total function mapping events to timestamps at which they are executed. We focus initially on a sequential schedule Σ_s which is equivalent to executing $\mathfrak{P}$ on a mono-processor architecture. The schedule assigns a unique timestamp to each event. Executing that schedule on any platform will yield the observation of $\sigma^o_{s,\pi}$ which is the same as Σ_s. Note that π specifies the execution platform characteristics.

Dually, for the provided parallel schedule $\Sigma_{\|}$ of program $\mathfrak{P}$ we can observe different serializations of events. As illustrated in the bottom of Figure 1 $\Sigma_{\|}$ assigns the same timestamp to different events. That is, those events were executed simultaneously by different processors or cores. The order in which parallel memory accesses are serialized depend on the execution platform. Most execution platforms allow relaxations on the order memory operations are executed, e.g. a read operation may be executed prior to a preceding write operation to a different memory location.

Thus, each thread may observe a different serialization of events. Though, its worth noting that the serialization observed by each thread may vary only within a bounded time slice we designate as δ. For all events prior to δ all threads consent on their read value.

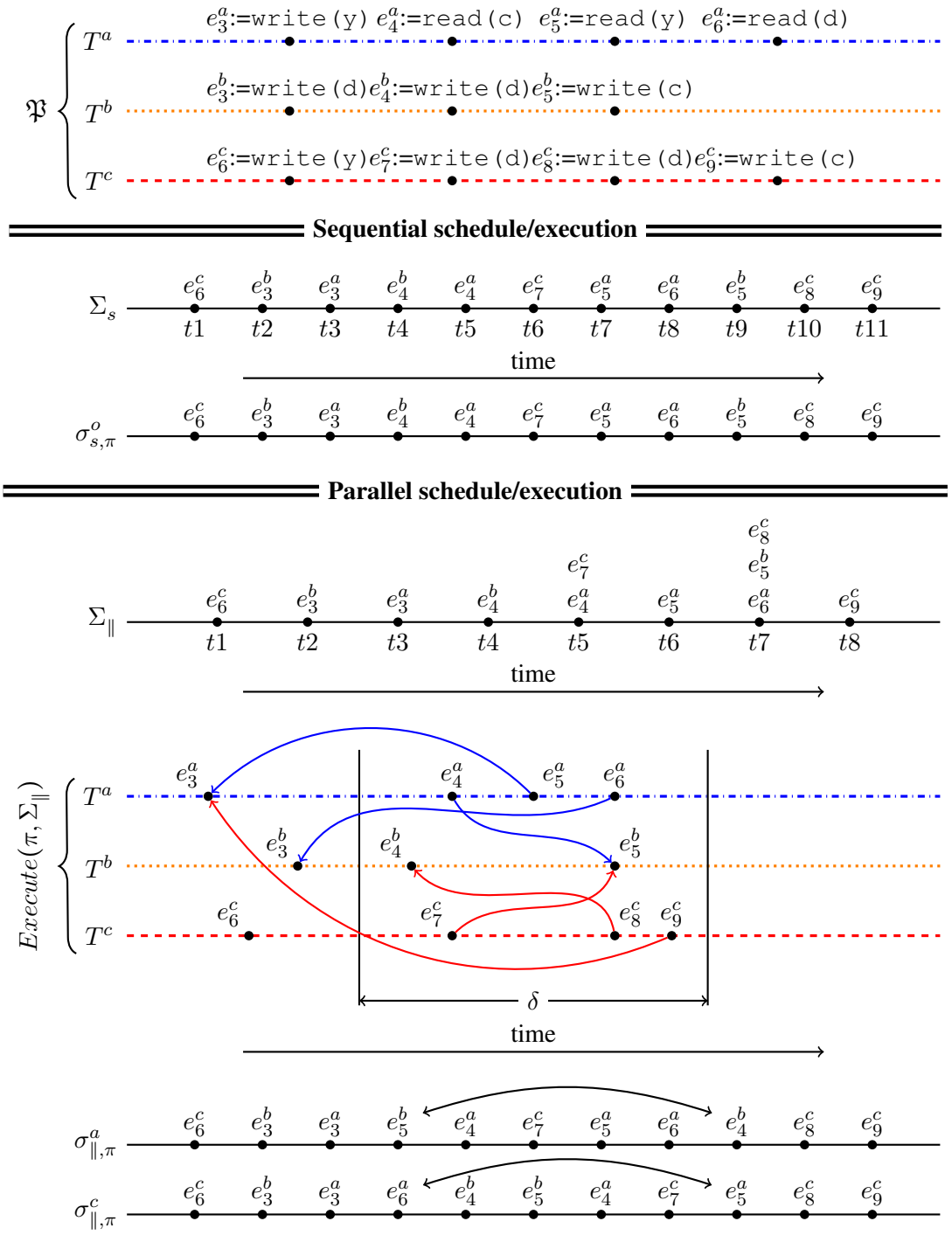


Figure 1: Obtaining serializations for a multithreaded program $\mathfrak{P}$

1.2 Information flow

The importance of information is invaluable and a number of security properties such as (i) confidentiality (ii) integrity and (iii) availability must be guaranteed. Nowadays information is stored, processed and transfered among a multitude of devices. Although encryption algorithms and communication protocols can ensure integrity and confidentiality, of data storage and transfer, they are not always used properly. Data are often stored un-encrypted and their security relies entirely on the operating system access control which is insufficient.

Moreover, benign software may unintentionally leak confidential data, or contain *vulnerabilities* that could be exploited by malicious software (malware). Malwares come in many flavors, viruses, worms, trojan horses, spywares etc. and focus in disclosing confidential information, or causing denial of services (e.g. crash programs) or capture the host computer (zombie computer) in order to accomplish further malicious tasks. To protect against software vulnerabilities *information flow* analysis is mandatory.

Information flow analyses trace how data processed by a program transit inside memory at execution time. Two popular analyses focusing on preserving security are *non-interference* and *taint* analysis. *Non-interference* focuses on confidentiality. It ensures that *high* (confidential) data do not flow into *low* (public) data. A program is safe if the same outputs are observed for different values of *high* data. *Taint* analysis on the other hand tracks untrusted data such as user or network input and checks how they influence vulnerable statements (e.g. return address of functions). Taint analysis was originally implemented in the *Perl* language for identifying security risks on web sites such as *SQL injections* and *buffer overflow* attacks. The term taint analysis is now widely used as a synonym to *dynamic information flow tracing* (DIFT). Moreover, the type of information traced is not restricted to untrusted data and thus more general analyses can be implemented.

1.3 Program validation

Testing has become an indispensable part of software development. It is applied to validate if a program meets its functional and extra-functional requirements for a specific (specially guided) execution. Testing is applied dynamically, that is by executing a program or fragment of a program and observe the execution for the tested property. We must note that testing is not exhaustive and thus it cannot be used for verification. In order to increase the confidence of programs, stress testing is applied to increase coverage of tested executions. If any functional errors (bugs) are found during testing they should be resolved through the debugging process.

Debugging of programs consists in finding errors (bugs) and correcting them accordingly. In order to debug an error it should be *reproducible* so that it can be replayed as many times necessary to detect its source. Several tools called debuggers help developers in this process. They allow a step-by-step execution of the program and inspection of memory at any point.

The non-determinism of concurrent program executions makes information flow tracking, testing and debugging more challenging. Concurrent modifications to shared memory affect the propagation of taintness and observations of the testing. Taint propagation or verdict of tests can be different for repeated executions using the same inputs. The non-deterministic execution is affected by many factors such as scheduling decisions taken by the operating system or even the serializa-

tion of events by the execution platform. Reproducing a non-deterministic execution precisely is nearly impossible. Thus, we can no longer specify precisely the execution that was tested, nor can we easily reproduce bugs in order to fix them. A partial solution to checking non-deterministic executions is *runtime prediction*. The intuition behind it is to infer/predict plausible serializations based on a concrete execution of the observed application. Thus, upon a single test execution we predict several neighboring executions which would be hard to enforce.

1.4 Our contribution

In this work we propose a predictive information flow analysis for parallel programs. Our analysis allows the parallel execution of applications which is the input to our prediction algorithm. The prediction is applied to bounded portions of code that were executed "simultaneously". Predictions are inferred by the iterative algorithm we propose. The prediction focuses into capturing information flows produced by plausible serializations of the parallel execution observed. The predictions should take into account the characteristics of the execution platform. We use *taint* analysis as a representative information flow analysis and consider sequentially consistent platforms. A considerable effort has been done to reduce false predictions. The contributions of this work are summarized as follows:

- Proposition of a runtime prediction technique for parallel executions.
- Definition of an algorithm for precise predictive taint analysis. The precision of predictions comes from:
  - (i) taking into account the underlaying memory model;
  - (ii) safely un-tainting memory locations;
  - (iii) respecting semantics of synchronization mechanisms (locks).
- The algorithm we propose avoids the enumeration of all serializations.
- Implementation of a proof of concept tool.

1.5 Organization of the document

We detail our work in the following sections. In section 2 we present a small survey on *taint* analysis and focus on its adaptation to parallel executions of multithreaded programs. Next, in section 3 we present our prediction technique, for identifying plausible serializations of a parallel execution. In section 4 we implicit the analysis for sequentially consistent memory model, to which we also add restrictions to respect un-tainting of memory locations and semantic of locks. Finally, we present some experimental results in section 5 and conclude in section 6.

2 Information flow analysis for multithreaded programs

Information flow analyses infer the data and control dependencies that can occur in a program. In software security such information is useful for detecting and preventing the exploit of software vulnerabilities and confidentiality breaches. In debugging and testing it can be useful for understanding how errors occur and what are their sources. Moreover, parallelizing compilers can also benefit of it since accesses to independent data can be safely parallelized. Tracing information flow in sequential programs is difficult due to dynamic memory allocations, control flow branchings etc. Adapting information flow analyses to multithreaded programs is even more challenging due to the non-deterministic execution, caused by the scheduling of threads and the relaxations of the execution platform.

Both *static* and *dynamic* techniques have been proposed to address the information flow tracing problem. *Static* approaches usually reason on source code level. A vast majority reposes on *type systems* to define languages that guarantee by construction secure information flows, i.e. executions that do not leak any confidential information. Volpano [VS97] and Sabelfeld [SM06] have proposed such sequential languages while Barthe [BRRS10] and Smith [SV98] include in their languages some basic primitives for multithreaded development. A drawback of these approaches is that they can reject programs that occasionally flow sensitive data. For instance a benign program may leak sensitive information only when sending a crash report to its developers. *Dynamic* approaches e.g. TaintEraser [ZJS+11] are better adapted since they monitor at runtime the flow of sensitive data and can interfere in order to prevent the leaking. Dynamic information flow tracing (DIFT) or taint analysis is widely used for detecting software vulnerabilities and avoid their exploit. As it is applied dynamically it is much more precise than static analyses. We detail taint analysis and how it propagates in section 2.1.

Hereafter we motivate taint analysis and give an overview of the techniques employed to address the problem. Further we introduce runtime prediction a method to generalize executions of multithreaded programs. We present our algorithm for predictive taint analysis. The implementation of our algorithm along with a proof of concept experimentation are presented in section 5.

2.1 Taint analysis

Taint analysis is a dynamic information flow tracing technique which consists of tainting (marking) sensitive or untrusted data and tracing their flow through a program. To perform taint analysis we need to specify: (i) the taint sources and (ii) a propagation policy of taintness. Often, untrusted data such as user input and network traffic are used as taint sources. The propagation of taintness may occur *explicitly* through copy of value (e.g. assignment) or *implicitly* through covert channels (e.g. control flow).

To limit propagation of taintness, a dual process of untainting is used to mark data as safe. This occurs by assigning an untainted value to data or by sanitizing it i.e. check they respect some rules and if necessary modify them such that they conform to these rules. We introduce the following notation for abstracting taint sources and sanitization:

T stands for $Taint$ and is used to abstract all possible taint sources. For instance, user input obtained through `scanf` function will be replaced by an assignment of T in the variable

written as in the example:

$$\texttt{scanf("\%d",val);} \Longrightarrow \texttt{val} = T;$$

U stands for $Untaint$ and is used to abstract sanitization functions. Sanitization is often used on untrusted data in order to ensure they are harmless and thus they can safely be untainted after its completion. Assuming function `clean_search` sanitizes a search string for SQL injections then we can make the following replacement:

$$\texttt{clean\_search(input);} \Longrightarrow \texttt{input} = U;$$

2.1.1 Explicit information flow

Listing 1 presents an excerpt of code where initially variable `a` gets tainted at instruction 2 (by reading user input into it). Also variable `e` gets eventually tainted through the dependency path `e` $\Rightarrow$ `d` $\Rightarrow$ `a` $\Rightarrow T$. The propagation of taintness in variable `d` is straight forward since we have an explicit copy of the tainted value. In the case of variable `b` the effect of the assignment is subtle to the taint propagation policy chosen. For instance, it may be assumed that merging tainted data with untainted absorbs the tainting effect. Most often though it suffices one operand to be tainted in order to propagate taint. Thus, in most existing taint analyses `b` would be considered tainted too. Finally, we note the sanitation of `a`.

2.1.2 Implicit information flow

Listing 2 presents an implicit information flow. Again variable `a` gets initially tainted. The tainted data controls program execution and thus information about it can leak. In this example, an external observer can infer information about the value of `a` by looking at the printed value of `b`. Thus information about `a` is implicitly propagated to all variables set inside the scope of control (this includes `c`). Variable `d` is not tainted since it is not affected by the value of `a`. Implicit flows are very hard to detect since covert channels can be implemented in many ways. Some typical examples are timing and storage channels.

```
1 int a,b,c,d,e;
2 a = T; // scanf("%d",a);
3 c = 21;
4 d = a;
5 b = c + a;
6 a = U; // sanitize(a);
7 e = d;
```

Listing 1: Explicit flow

```
1 int a,b,c,d;
2 a = T; //scanf("%d",a);
3 if( a >10 ){
4   b = 1;
5 }else{
6   b = 0;
7   c = 2;
8 }
9 d = 10;
10 printf("%d",b);
```

Listing 2: Implicit flow

Because implicit flows are based on covert channels they are tedious to track both statically and dynamically and are often neglected. Implicit information flows are mostly critical for the non-interference property where confidential data can leak unconsciously. In the context of taint analysis and vulnerability detection implicit flows affect subtly the exploitation of a vulnerability. Moreover, implicit propagation of taint introduces too many false positives which degrades the efficiency of the analysis.

2.1.3 Application of taint analysis

As mentioned earlier taint analysis targets mostly vulnerability detection and prevention. Therefore, most taint analyses are performed dynamically at runtime. Tainted data are tracked down and the appropriate checks are performed, when necessary, in order to respect the security property.

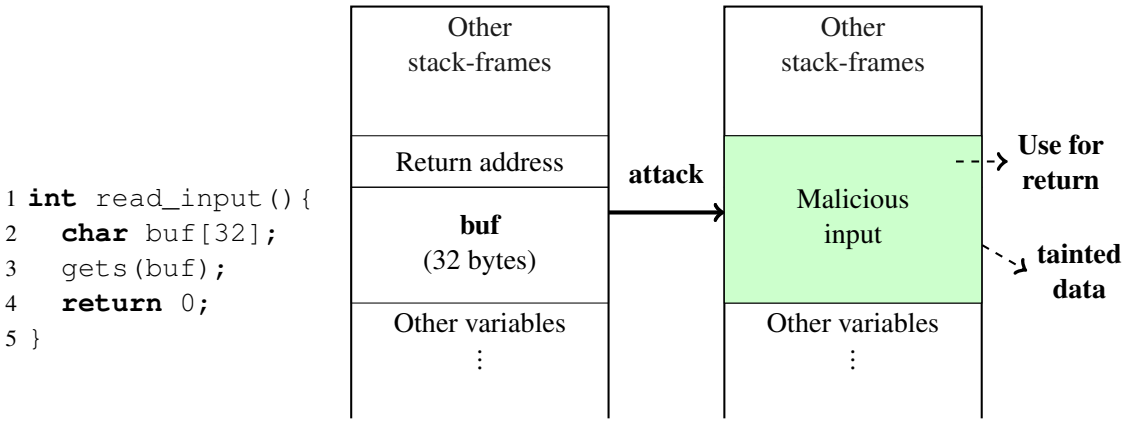


Figure 2: Stack smashing by buffer overflow.

A typical example demonstrating how taint analysis is used for vulnerability prevention is stack smashing caused by a buffer overflow. Figure 2 describes such an attack. On the left of the figure we present the code of the function which reads user input into a buffer of size 32 bytes. The programmer assumes the size associated to the buffer is enough to hold the input provided. If a user enters a longer input then the data will overflow the buffer resulting into writing a new value to the return address associated to this stack frame (illustrated on the right stack frame). This vulnerability can be exploited by ensuring that the return address is not overwritten with random data but with an address to a malicious set of instructions. To prevent that from happening a taint analysis should check if the return address is tainted prior to jumping to it.

2.2 Tracing taintness

Dynamic analysis techniques are widely used in the context of multithreaded applications for runtime error detection like deadlocks ([LELS05, CFC12]) and data races ([SBN+97, SI09]. 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 ([BKK10, ESKK08]) and enforcement of security policies ([ZJS+11, CM09]).

Building dynamic analysis tools necessitates integrating some monitoring facilities to the analyzed application. Monitoring features are added either at source code level or binary level, either statically or dynamically. Waddington et al. [WRS] present a survey on these techniques. Working at the binary level allows to analyze programs for which source code is not available such as malwares or libraries. A major drawback is that high level information is lost making it harder to reason about the program.

Instrumentation code is often added statically in applications as explicit 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 binary rewriting functionality of frameworks like Dyninst [BH00]. Hereafter we take a closer look to dynamic binary instrumentation (DBI) techniques since they are the most widely used.

2.2.1 Dynamic binary instrumentation

In general, DBI frameworks ([NS07, BH00, LCM+05]) 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 [NS07] 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 [LCM+05] 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. [HLC09] point out the difficulties in implementing a framework that scales well in a paral-

lel 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.

2.2.2 Sequential taint analysis

All taint analyzes are decomposed into three distinct phases:(i) tainting (ii) tracing and (iii) asserting. The first two were presented earlier in section 2.1 and consist in defining what data are tainted and how taintness propagates. The third phase consists into checking how tainted data are used. The property to be asserted affects the first two phases too.

Often taint analyzes implemented with DBI [NS05, ZCYH05, CZYH06, QWL+06, ZJS+11, GLG12] focus on the same properties such as buffer overflows, format string attacks, stack smashing etc. and compare with each other in terms of precision and performance. The major overheads in these analyzes are caused by *instrumentation* and *updating shadow memory*.

Shadow memories are used to store information about taintness. A mapping exists between the registers and address space of the application to the shadow memory, as illustrated in Figure 3. For performance and memory usage optimization shadow memories are usually implemented as bitvectors. Each bit indicates whether the mapped memory is tainted or not. The granularity of the mapping may vary, but most often a bit corresponds to a byte of address space or register. Because the lookup and updating of shadow memory occurs practically for each instruction executed by a processor Nagarajan and Gupta [NG09] propose architectural support for their implementation. Their proposal focuses on multiprocessors and thus incurs modifications both at the instruction set and at the cache coherency protocols.

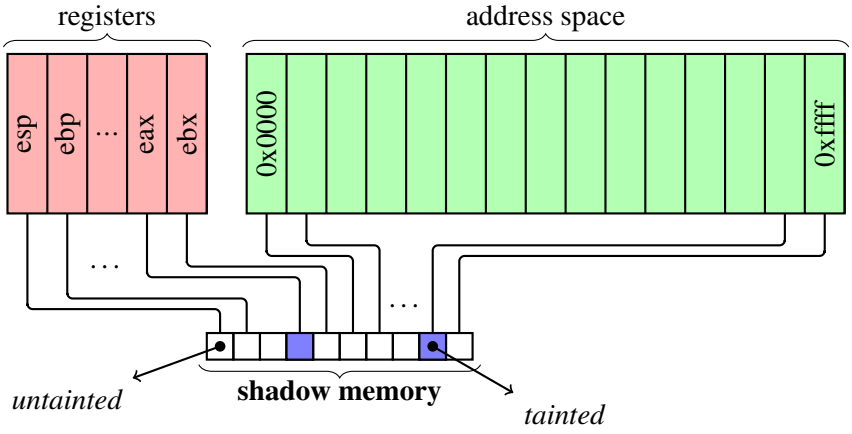


Figure 3: Shadow memory mapping.

Newsome and Dong proposed TaintCheck [NS05] for detecting exploits on commodity software and produce signatures for their early detection and avoidance. They used Valgring [NS07] for instrumentation which penalizes the monitored execution due to its emulation approach. Their information flow tracing is limited to *move* (e.g. load, store, push, pop) and *arithmetic* (e.g. add, sub, xor) instructions. During arithmetic operations some special registers are updated called EFLAGS which are not taken into account. For shadow memory they map each byte of memory

(register or address space) to a four-byte pointer, linked to either a tainted data structure [1] or to NULL depending on its taint status. Shadow memory is kept in a page-table like structure in order to reduce its size. The assert phase checks if tainted data are used in jump instructions or passed as arguments to system calls.

Further to taint propagation TaintCheck keeps a log allowing to trace the flow from the source that tainted it down to the exploit position. This is necessary for generating the signature of the attack. Moreover, once an exploit is detected the programs execution may continue under a constrained environment which allows to learn what is the goal of the attack. This information is useful for undoing, if possible, the damage done by a malware.

Zhao et al. present DOG [ZCYH05] a program monitoring framework built on top of Dynamorio [Bru04] for detecting exploits but also preventing confidentiality leaks. DOG provides a graphical interface from which one can define the taint sources, and associate to each source a propagation policy and a set of assertions and actions to perform if an exploit is detected. The propagation of taint supported is similar to TaintCheck [NS05] apart that they take into account the EFLAGS and allow for implicit propagation through control. Because implicit propagation can introduce many false positives DOG allows the user to specify regions in the program where it can be applied. To optimize taint tracing a bit-vector is used. Each byte corresponds to a bit with value 1 marking it as tainted and 0 as untainted. Moreover, they do not use a page-table based strategy as in TaintCheck but instead they devise a mapping where it suffices to add a *shadow_base* to the address to locate its mapping. The taint checks provided by DOG are somehow typical, i.e. format string attacks, stack smashing, etc.

Dytan [CLO07] is yet another framework for taint analysis. It is implemented using Pin [LCM+05] and as DOG it supports both implicit and explicit flow propagation. In addition to a simple XML configuration the framework also provides an easily extendable interface allowing the rapid development of more elaborated taint analyzes. The taint information is also stored in bit-vectors and the granularity of memory mapped is one byte.

A most recent work TaintEraser [ZJS+11] focuses on confidentiality, it blocks unintended data exposure to the network or local file system by applications. TaintEraser makes several optimizations in taint analysis without losing in precision. First, it uses *function summaries* which resume the effects of a functions execution and thus there is no need to instrument it at runtime. Moreover, they perform on-demand instrumentation, i.e. they do not instrument the entire program execution. Finally, to enforce confidentiality of sensitive data it allows to log the leak, block the action or replace the sensitive data with random ones. The output channels protected are network connections and files.

All frameworks presented above use DBI to add monitoring. Figure 4 illustrates an overview of their mode of operation. The DBI framework observes the instructions executed by the processing unit and updates accordingly the shadow memory and takes action if needed. As presented in the figure, the execution of multithreaded programs is serialized. This is convenient for monitoring since the DBI frameworks observe a sequential schedule Σ_s which allows the shadow memory to be updated precisely.

Adding instrumentation at runtime incurs two drawbacks. First, it penalizes execution having

[1] this is similar to the taint object T we defined, its a fixed point for taintness

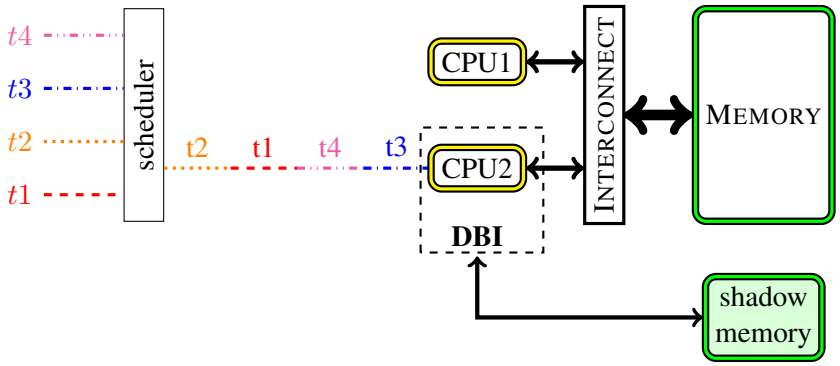


Figure 4: DIFT analysis using Dynamic Binary Instrumentation frameworks

to produce the instrumentation dynamically, at least for the first [1] time they get executed. Second, it is hard to apply any optimizations since high level program structure has been lost. Saxena et al. [SSP08] try to weaken these problems by proposing a binary rewriting technique for adding instrumentation. Prior to adding the monitoring code they extract as much high level information as possible from x86 executables, so that optimizations can be applied.

2.2.3 Optimizing DIFT

A great challenge DBI frameworks are facing is dealing efficiently and correctly with multi-threaded programs and their parallel execution achieved on the multicore platforms. As illustrated in Figure 4 existing DIFT analyses based DBI frameworks serialize their execution. Though necessary for tracing information flow precisely it incurs a great penalization of the execution time. To improve DIFT analyzes several solutions requiring the support of specialized hardware have been proposed.

Nagarajan et al. [NKWG08] takes advantage of multicore processors to perform DIFT transparently and efficiently. Their solution reposes on spawning a new thread dedicated to the analysis and running on a dedicated core in parallel with the main thread (the monitored application). The monitoring core tracks taintness and sends an interrupt to the main thread when the use of a tainted value violates the specified security policy. Intense communication between the cores executing main and monitor thread respectively is required. Initially shared memory was used for their communication but it added too much overhead to the execution. Thus, they proposed the usage of a dedicated hardware FIFO [2] buffer. Although this buffer does not exist in current multicore processors, it has been proposed by several other works [RVS+06, SKSP06]. For their experimentations they used the *Simics* full system simulator on which they implemented the hardware FIFO queue. The results they obtained showed a 48% overhead which is much better than aforementioned frameworks which introduced an overhead of about 300% and more.

The work of Ruwase et al. [RGM+08] reposes on the *log-based architecture* (LBA [CKS+08]) to implement a parallel dynamic information flow analysis. LBA introduces several hardware components in the CPU design that allow the extraction of a log trace for a monitored application. The log can be read by the monitoring thread. The analysis of the log happens in parallel. It is

[1] instrumented code is stored into code caches for later use
[2] First In First Out or queue like storage and processing policy

broke into segments each processed by a worker thread running on a dedicated core. The worker threads create summaries of segments and send them to the monitoring/master thread which updates meta-data and makes the appropriate checks. For the parallelized DIFT they proposed a big number of worker threads is necessary, but it does not always guarantee a speedup compared to sequential frameworks.

A most recent work by Ozsoy et al. proposes SIFT [OPAGS11] which takes advantage of symmetric multithreading (SMT). The implementation of their work though necessitates modifications which increase the size of the processor core by core by 4.5%. Although it is relatively small compared to the solution proposed in Raksha [DKK07] which increases chip size by 20%. Hardware-based DIFT solutions seem very appealing but necessitate non-trivial hardware modifications which make the design of processing units more complex. Thus chip manufacturers are not willing into adapting them.

2.3 Extending monitored traces

Dynamic information flow analysis performed either at software level, using a DBI framework, or with support of sophisticated hardware, allow the meticulous analysis of a single execution trace. That is, the verdict concerns only the specific execution which is often serialized. Such a solution is very intrusive and hides sources of concurrency bugs such as races caused by non-deterministic scheduling and effects of weak memory model relaxations.

Although monitoring of applications is useful itself, as long as the performance losses are acceptable, in some contexts such as debugging or testing, limiting the verdict to a single execution is too restrictive. To overcome this problem *runtime prediction* is used to expand the analysis by inferring executions. The inferred executions capture different interleavings for the executed application.

Depending on the accuracy of the interleaving computation the prediction may under-approximate (miss errors) or over-approximate (produce false positives). In the former case, the initial execution trace is usually captured as a totally ordered sequence of events which is relaxed pessimistically i.e. allowing only a subset of feasible interleavings. In the latter case, execution traces are conceived as unordered sets of events and interleavings are computed by enumerating all possible interleavings and then eliminating some unfeasible paths (e.g. based on happens before relations).

2.3.1 Runtime prediction for concurrency bugs

Runtime prediction has been widely used in the identification of concurrency bugs such as race conditions and deadlocks. To perform such analysis the frameworks proposed in the literature [JNPS09, SFM10, WG12] abstract executions by logging information necessary to discover interleavings susceptible to cause concurrency bugs. The logs consist of shared memory accesses and various synchronization primitives such as lock acquisitions and releases, thread creation and join etc. The frameworks are differentiated by the logged information, the algorithms detecting interleavings and either they over or under approximate. The algorithms used can be split into *enumerative* and *symbolic*. In the former case all interleavings are enumerated and then bogus ones

are filtered, while in the latter case constraints on interleavings are encoded into logic formulas fed to SMT [1] solvers [DM06].

Several works use enumerative algorithms. Some of them [WS06b, WS06a] over-approximate since they solely rely on the algorithms inferring the interleavings. To reduce false positives CalFuzzer [JNPS09] and PENELOPE [SFM10] try to infer a schedule capable to exhibit the concurrency bug. The inferred schedule is executed and if the bug occurs then it is reported by the framework, else it is dropped.

In the symbolic category Wang et al. [WG12] provide a detailed survey. Moreover they briefly present their contribution in the domain. First, they mention a theoretical optimal solution they proposed, the CTP [WCGY09] (Concurrent Trace Program), which captures all interleavings that can possibly be inferred from a single trace, without introducing any bogus interleavings. Subsequently they present two abstractions UCG [KW10] (Universal Causality Graph) which over-approximates and a dual work, TSA [SMWG11] which under-approximates the set of traces computed in TCP.

2.3.2 Runtime prediction applied to information flow

In the context of information flow *runtime prediction* has not yet been widely used. We present hereafter two recent analyses: DTAM [GLG12] and Butterfly [GVC+10].

In their work Ganai et al. [GLG12] propose DTAM analysis which identifies a subset of tainted input sources and shared objects that can affect the execution of a multithreaded program. That is, the tainted data have an impact on the control-flow of the program or its shared state. The tainted data get classified according to six relevancy types that describe how they tainted data can affect the program execution. To infer the information flow dependencies DTAM proposes a serial variation DTAM_{serial} and two parallel ones $\text{DTAM}_{parallel}$ and DTAM_{hybrid}.

DTAM_{serial} monitors the serialized execution of the multithreaded program and keeps track of taintness as in usual DIFT analyses. In the parallel variations each thread performs thread-local taint propagation. The information flow between threads is taken into account during the offline phase. For the offline phase relevant information needs to be logged. Each thread logs shared memory accesses along with the runtime taint value they have computed. Some basic synchronization primitives are also kept into the log such as fork/joins and wait/notifies allowing to infer happens before relations. For their relevancy analysis even conditionals are logged.

The difference between the $\text{DTAM}_{parallel}$ and DTAM_{hybrid} is that $\text{DTAM}_{parallel}$ does not take into account happens before relations and thus the results are less accurate. Else, the inter-thread propagation in both cases is rather coarse. Once a shared memory location is tainted in a thread, it remains indefinitely and propagates to all other threads. Such an approach drastically over-taints and it can result into considering everything as tainted.

The main objective of Goodstein et al. [GVC+10] 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

[1] Satisfiability Modulo Theories

the lifeguard reasoning about the *set* of parallel executions. To solve this issue, the authors considered (monitored) executions produced on specific machine architectures [CKS+08] on which *heartbeats* can be sent regularly as *synchronization 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.

2.3.3 Positioning of our work

Our work is inspired from Butterfly analysis [GVC+10] though the 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 analyzed 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 of view, this makes the algorithms simpler, without sacrificing efficiency. Finally, a further contribution is that we take into account lock-set information to reduce the number of false positives.

3 Predictive explicit taint analysis

Hereafter we present our approach to *predictive explicit taint analysis* of multithreaded programs. The motivation is to use it for test validation, that is extend the results of a tested parallel execution to the set of plausible serializations that could have occurred. Since we are in a testing context our predictions do not need to be sound (taint value can be over or under approximated). For a test to be representative of a concrete execution the monitoring should be as transparent as possible. As presented in section 2.2.2 most works force the serialization of multithreaded applications, which is very intrusive. We do not impose such restrictions to the scheduling, i.e. we allow the parallel execution of the application, and reason a posteriori about taint propagations.

3.1 Overview of our approach

In our approach, an abstract view of which is presented in Figure 5, we propose an *offline sliding window-based analysis*. First, the multithreaded application is executed and a parallel schedule $\Sigma_{||}$ is captured in the form of log files. A log file is recorded per executed thread containing the timestamped sequence of events produced by the thread mapped to it (upper part of Figure 5).

Next, the log files are sliced into so called *epochs* and the sliding window-based taint analysis is applied (lower part of Figure 5).

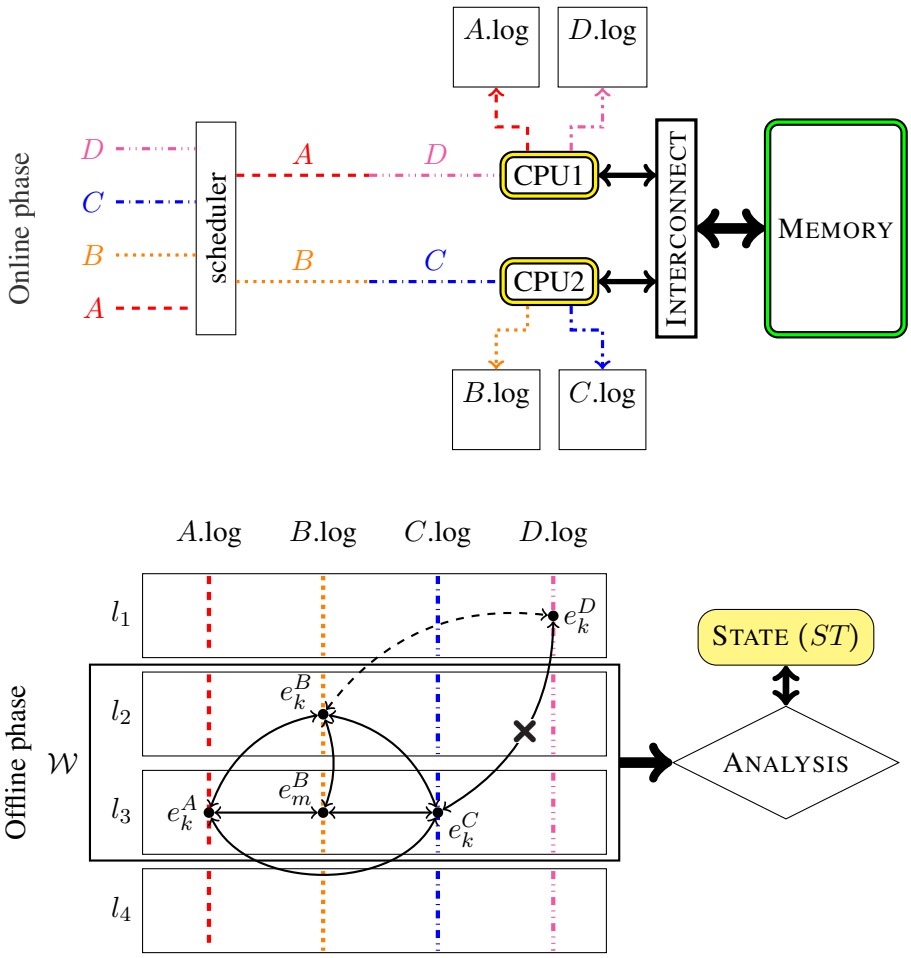


Figure 5: Overview of our approach

Due to the information flow property we are interested in (taint propagation), the logging of events is exhaustive. Typically, all memory accesses, affecting both *shared* and *thread-local* variables, in the form of *use/def* relations and some synchronization events. We remind the T and U notations introduced in section 2.1 where T is a fixed tainted variable and dually U an untainted one. For an event e we introduce the following functions:

$Def(e)$	returns the singleton set of variables *defined* i.e. written by event e
$Used(e)$	returns the set of variables *used* i.e. read by event e

Computing all interleavings (i.e. all serializations) of logged events is impractical. To avoid the interleaving explosion problem we propose (i) the slicing of logs into epochs (l) which limits the number of events to interleave and (ii) a processing algorithm which infers taint propagation without enumerating all serializations. Because the slicing can occur between arbitrary events there is no guarantee that a happens before relation is established for events belonging to consecutive epochs. Thus, we extend the bounding of interleavings to events belonging in a window consisting of two adjacent epochs. Section 3.2 provides more details on slicing.

The lower part of Figure 5 illustrates a window consisting of two consecutive epochs ($\mathcal{W}$=$\{l_2$,$l_3\}$) and the considered interleavings of events e_k^A, e_k^B, e_m^B, e_k^C in it. We note that also events belonging to the same thread can be interleaved. This allows to reason on taint propagation under relaxed memory models. Moreover the figure presents how slicing bounds the prediction to events belonging to adjacent epochs only. For instance, the interleaving between events e_k^B and e_k^D denoted with a dashed line is considered in the preceding window consisting of epochs l_1, l_2. On the contrary the interleaving between e_k^C and e_k^D is crossed out because it will not be considered by the analysis since the events do not belong to adjacent epochs. Similarly, the interleaving of events e_k^A and e_m^B with e_k^D are not taken into account.

We apply our analysis using a sliding window consisting of two adjacent epochs. The window slides over epochs thus all interleavings of an event with events in its preceding and succeeding epochs are explored. The analysis identifies taint propagations inside the currently analyzed window $\mathcal{W}$ and summarizes their effect in state ST, which acts as a *shadow memory*.

3.2 Slicing the parallel schedule $\Sigma_{\|}$(log files)

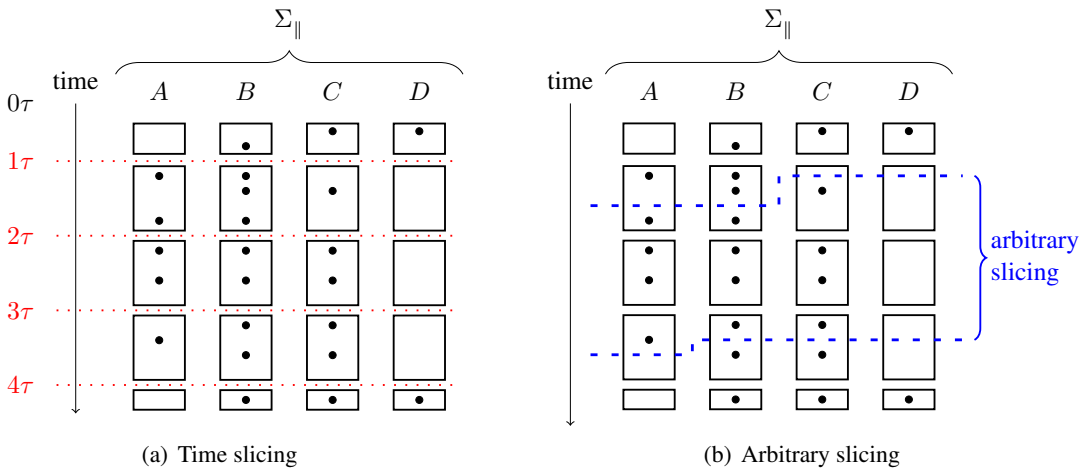
The slicing of log files into epochs affects the prediction since it defines which events can be interleaved. The slicing technique we use is *time-based*, that is we define a time period τ which slices the logs as illustrated in Figure 6(a). The time slicing is well adapted for our purpose because it allows the analysis to consider interleavings of events that were executed simultaneously, or at least in parallel with respect to the chosen period τ.

Choosing the value of τ is delicate. In principle it should be large enough to capture (i) the delta between the execution of an event and the assignment of the timestamp and (ii) the effects of the platform on the ordering of executed instructions (weak memory models). Taking into account criterion (ii) is meaningful only when the logging of events is at the assembly level and the timestamping utterly precise. We note that, by setting a large value for τ the analysis may infer taint propagations caused by different schedules. Dually, choosing a small value will under-approximate taint propagation, and thus the analysis will not infer taintness for all feasible serializations under the observed schedule. Finally, if the entire execution log is split into just two epochs (i.e. one window) then the analysis reasons about all possible executions of the program.

In general, the slicing can be performed arbitrarily. Figure 6(b) illustrates such an arbitrary slicing delimited by thick loosely dashed lines. Some heuristics that can produce interesting slices are to use context switches or synchronization barriers as the slicing points. In the case of synchronization barriers for instance, slicing at these points is not sufficient. A dummy epoch should be introduced such that it forces the analysis not to reason about the interleavings. The dummy epoch should define empty blocks for the threads concerned by the synchronization.

We introduce hereafter some key notations that we use in the sequel. As mentioned previously the slicing of log files defines epochs as illustrated in Figure 7. The events of a thread belonging to an epoch form a *block*. Each block is uniquely identified by a tuple (l, t) where l is the epoch it resides in and t is the thread identifier. Events within a block are uniquely identified by a triplet (l, t, i) where l, t specify the block it belongs to, and i is the identifier of the event. As illustrated in Figure 7 event $e_i^B = (l, B, i)$.

We further define the functions $Thr(e)$ and $Epoch(e)$ which return respectively the thread that



(a) Time slicing (b) Arbitrary slicing

Figure 6: Slicing $\Sigma_{\|}$ into epochs

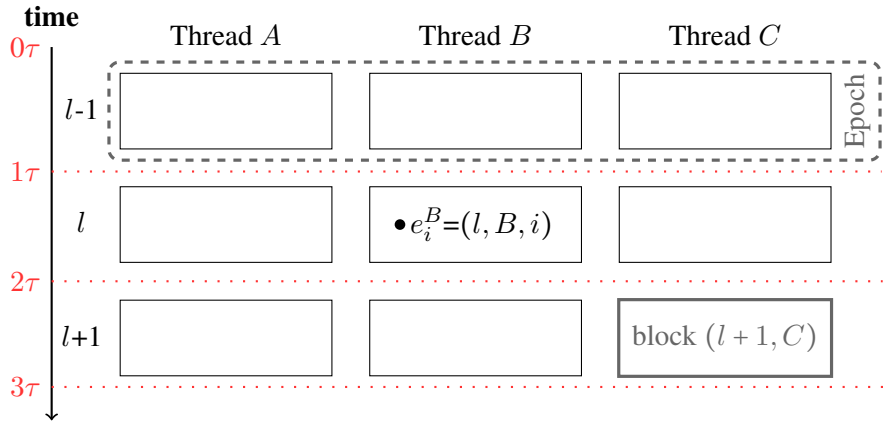


Figure 7: Basic notations

executed event e and the epoch it belongs to. Finally, we introduce a binary reflexive operator $\nleftrightarrow$ which denotes two events can be interleaved based on the window interleaving assumption. More formally:

$$e_k \nleftrightarrow e_m \Rightarrow |Epoch(e_k) - Epoch(e_m)| \leq 1$$

3.3 Sliding window-based explicit taint propagation

As mentioned earlier what we propose is an *offline sliding window-based analysis* for predicting explicit taint propagation. There are two aspects in our analysis:

(i) prediction of explicit taint propagation within a window, and summarization of its effects;

(ii) reasoning correctly about the sliding windows which causes them to overlap.

We introduce hereafter the notations used in our sliding window analysis. Figure 8 illustrates two consecutive windows $\mathcal{W}' = \{l_h, l_b\}$ and $\mathcal{W} = \{l_b, l_t\}$ where $\mathcal{W}$ is the *currently analyzed*

window consisting of epochs labeled l_b, l_t while $\mathcal{W}'$ is the preceding window consisting of epochs labeled l_h, l_b. The labeling of epochs is relative to the currently analyzed window and is adopted from [GVC+10]. The upper epoch of the currently analyzed window ($\mathcal{W}$ in Figure 8) is called *body* (l_b) while the lower one *tail* (l_t), finally the epoch preceding body is called *head* (l_h). We remind that $ST_{\mathcal{W}'}$ summarizes the taint predictions down to the indexed window ($\mathcal{W}'$).

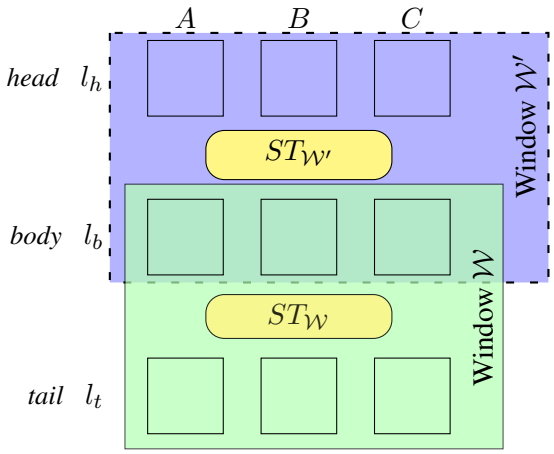


Figure 8: Sliding window based analysis

Prior to abording explicit taint prediction of a window we define explicit tainting and un-tainting of a variable. Next, we provide a formal definition of taintness on a serialized execution of events. Finally, we adapt this definition to fit a serialization of events in our window-based taint prediction.

We use the oracle $isTainted(x)$ which asserts if a variable x is tainted or not. It allows us to define explicit tainting/generation ($gen(e)$) and respectively un-tainting/killing ($kill(e)$) of the variable defined by an event e:

$$gen(e) = \begin{cases} \{Def(e)\} & \text{if } \exists x \in Used(e) \quad \text{s.t. } isTainted(x) \\ \{\varnothing\} & \text{if } \nexists x \in Used(e) \quad \text{s.t. } isTainted(x) \end{cases}$$

$$kill(e) = \begin{cases} \{Def(e)\} & \text{iff } \nexists x \in Used(e) \quad \text{s.t. } isTainted(x) \\ \{\varnothing\} & \text{if } \exists x \in Used(e) \quad \text{s.t. } isTainted(x) \end{cases}$$

Taint propagation occurs through a series of taintings over a serialization. That is, there is a tainting source that causes variables to be tainted. Moreover, a tainted variable remains so until some event un-taints it. We provide hereafter the definition of taintness for a variable x at an event e of a serialization σ.

Definition 3.1 (Taintness on a serialized execution: $taint(\sigma, x, e_k)$)
Let σ be a valid serialization of the analyzed application, x a variable, and e_k an event in σ. We (recursively) define predicate $taint(\sigma, e_k, x)$, meaning that variable x is tainted at event e_k on σ. Note that event indexes correspond to the position/order of events on the serialization.

$$taint(\sigma, e_k, x) \equiv \begin{cases} x = T \quad \bigvee \\ \exists m \leq k \textit{ such that:} \quad Def(e_m) = x \quad \wedge \\ \qquad\qquad\qquad\qquad \exists y \in Used(e_m) \,.\, taint(\sigma, e_m, y) \quad \wedge \\ \qquad\qquad\qquad\qquad \forall n \,.\, m < n < k \Rightarrow Def(e_n) \neq x \end{cases}$$

Intuitively, a variable x is tainted at event e_k of σ if it was assigned with a tainted variable at a preceding event e_m (or at the event e_k itself), and never re-assigned in between. Figure fig:def-taint illustrates the definition of $taint(\sigma, e_k, \mathtt{x})$. The serialization is not complete, since the occurrence of events post e_k do not affect the taint value of x at event e_k. Applying the taint predicate recursively, tracks back to the initial taint source which is always variable T. We remind that T is a constantly tainted variable which abstracts all taint sources.

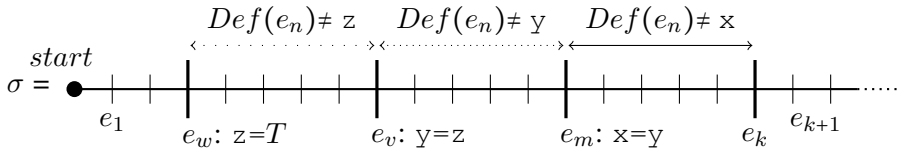


Figure 9: Taint definition for a concrete serialization

We adapt the taint definition above to our window-based prediction analysis. Note that, the definition is applied on a valid serialization $\sigma^i_{\mathcal{W}}$ of events in $\mathcal{W}$. Since the serialization is bounded to events belonging to the window $\mathcal{W}$, recursively applying the taint definition may not be able to reach the constantly tainted variable T. Thus, the taint definition must rely on summarizations of taint predictions, that is it suffices to reach a variable in $ST_{\mathcal{W}'}$. We provide the definition of *window-based taintness* ($taint\mathcal{W}(\sigma^i_{\mathcal{W}}, e_k, x)$) on a serialization $\sigma^i_{\mathcal{W}}$ of events in window $\mathcal{W}$.

Definition 3.2 (Taintness on a serialization of a window $\mathcal{W}$)
Let $\sigma^i_{\mathcal{W}}$ be a valid serialization of the currently analyzed window $\mathcal{W}$, x a variable, and e_k an event in $\sigma^i_{\mathcal{W}}$. We (recursively) define predicate $taint\mathcal{W}(\sigma^i_{\mathcal{W}}, e_k, x)$, meaning that variable x is tainted at event e_k on $\sigma^i_{\mathcal{W}}$.

$$taint\mathcal{W}(\sigma^i_{\mathcal{W}}, e_k, x) \equiv \begin{cases} x = T \quad \bigvee \\ x \in ST_{\mathcal{W}'} \wedge \nexists j < k \textit{such that: } Def(e_j) = x \quad \bigvee \\ \exists j \leq k \textit{ such that: } \quad Def(e_j) = x \quad \wedge \\ \qquad\qquad\qquad\qquad \exists y \in Used(e_j) \,.\, taint(\sigma^i_{\mathcal{W}}, e_j, y) \quad \wedge \\ \qquad\qquad\qquad\qquad \forall m \,.\, j < m < k \Rightarrow Def(e_m) \neq x \end{cases}$$

Figure 10 illustrates the application of definition $taint\mathcal{W}(\sigma^i_{\mathcal{W}}, e_k, x)$ on an example. The events preceding $\mathcal{W}$ are abstracted in the dotted path and the predictions of their plausible serializations, with respect to a given slicing and the application of window-based taint prediction to it down to $\mathcal{W}'$, are summarized in $ST_{\mathcal{W}'}$. Applying the definition for variable x at event e_k we obtain the recursive calls of events pointed by the arrows initiated from e_k. The recursion ends in the summary of the preceding window $ST_{\mathcal{W}'}$.

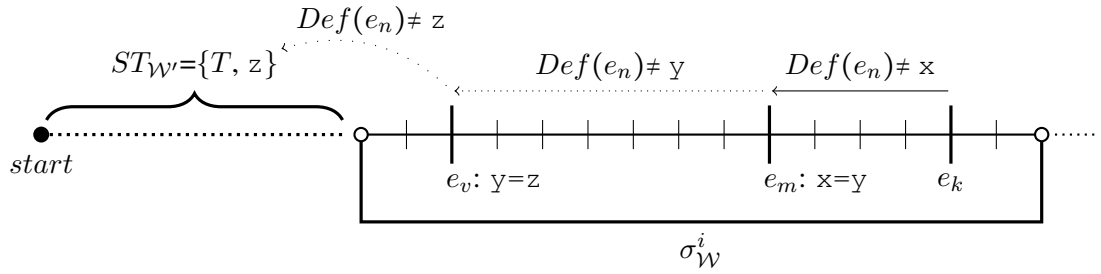


Figure 10: Taint definition for a plausible serialization of events in a window $\mathcal{W}$

3.4 Explicit taint prediction in a window

To introduce explicit taint prediction of a window $\mathcal{W}$ we consider the simple case where:

(i) events in $\mathcal{W}$ can arbitrarily interleave, even those produced by the same thread. That is, any serialization $\sigma^i_{\mathcal{W}}$ of events is considered valid (no memory model restrictions). In section 4 we illustrate how to enforce sequential consistency.

(ii) events that kill/un-taint variables are ignored. This assumption simplifies propagation of taintness and is often used e.g. [GLG12]. This assumption will be raised in the case of sequentially consistent serializations in section 4.2.

We provide hereafter the definition of *relaxed taintness* which introduces the ignoring of killing/un-tainting variables:

Definition 3.3 (Relaxed taintness on a serialization of a window $\mathcal{W}$)
Let $\sigma^i_{\mathcal{W}}$ be an arbitrary serialization of the currently analyzed window $\mathcal{W}$ ($\sigma^i_{\mathcal{W}} = \{(e_1, \ldots, e_n) \mid \forall k < m \Rightarrow e_k \nleftrightarrow e_m\}$), x a variable, and e_k an event in $\sigma^i_{\mathcal{W}}$. We (recursively) define predicate $taint\mathcal{R}(\sigma^i_{\mathcal{W}}, e_k, x)$, meaning that variable x is tainted at event e_k on $\sigma^i_{\mathcal{W}}$.

$$taint\mathcal{R}(\sigma^i_{\mathcal{W}}, e_k, x) \equiv \begin{cases} x = T \quad \bigvee \quad x \in ST_{\mathcal{W}'} \quad \bigvee \\ \exists j \leq k \text{ such that:} \quad Def(e_j) = x \quad \wedge \\ \qquad\qquad\qquad\quad \exists y \in Used(e_j)\,.\, taint\mathcal{R}(\sigma^i_{\mathcal{W}}, e_j, y) \end{cases}$$

Figure 11 presents the definition of relaxed taintness propagation in a window. As illustrated, the events killing variables are ignored i.e. the condition of not redefining a variable between its tainting and its usage to taint some other variable has been removed. For example, the effect of event e_w (which is crossed out) is ignored, thus it does not prevent the taint propagation at event e_m.

3.4.1 Enumerative approach

A natural way of predicting taint propagation in a window $\mathcal{W}$ is to compute all serializations $\sigma^i_{\mathcal{W}}$ by enumerating all permutations of events in it. Given the cardinality (number of events)

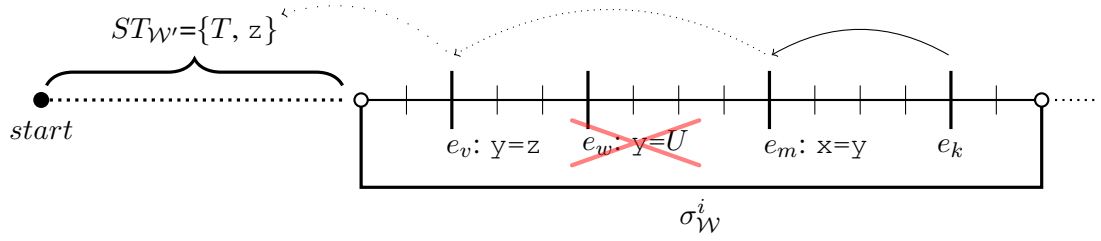


Figure 11: Relaxed taint definition for a plausible serialization of events in a window $\mathcal{W}$

of $\mathcal{W}$ denoted as $|\mathcal{W}|$, there will be $|\mathcal{W}|!$ such serializations, since all interleavings are feasible. A sequential taint analysis should then be applied on each serialization $\sigma^i_{\mathcal{W}}$, with $i \in [1, |\mathcal{W}|]$, using as initial state $ST_{\mathcal{W}'}$ and producing a local state $ST^i_{\mathcal{W}}$ which predicts *relaxed taint propagation* for the analyzed serialization. That is, for each variable x in $ST^i_{\mathcal{W}}$ the predicate $taint\mathcal{R}(\sigma^i_{\mathcal{W}}, last(\sigma^i_{\mathcal{W}}), x)$ holds, and likewise. By $last(\sigma^i_{\mathcal{W}})$ we denote the last event of serialization $\sigma^i_{\mathcal{W}}$.

$$x \in ST^i_{\mathcal{W}} \Leftrightarrow taint\mathcal{R}(\sigma^i_{\mathcal{W}}, last(\sigma^i_{\mathcal{W}}), x)$$

We present in Algorithm 1 how each plausible serialization of window $\mathcal{W}$ is analyzed. First, a copy of taint predictions down to the preceding window ($ST_{\mathcal{W}'}$) is made. The copy is updated locally such that when a variable gets tainted it is added to the state. Dually, when a variable is untainted no action is taken since these events are ignored.

Algorithm 1 Relaxed taint analysis of a serialization (kills are ignored)

In: $\sigma^i_{\mathcal{W}}, ST_{\mathcal{W}'}$
1: $ST^i_{\mathcal{W}} \leftarrow ST_{\mathcal{W}'}$
2: **for all** $e \in \sigma^i_{\mathcal{W}}$ **do**
3: **if** $Used(e) \cap ST^i_{\mathcal{W}} \neq \varnothing$ **then**
4: $ST^i_{\mathcal{W}} \leftarrow ST^i_{\mathcal{W}} \cup Def(e)$
5: **end if**
6: **end for**
Out: $ST^i_{\mathcal{W}}$

The analysis of each serialization $\sigma^i_{\mathcal{W}}$ with Algorithm 1 computes its *relaxed taintness* into $ST^i_{\mathcal{W}}$ which contains the set of variables that can be tainted by $\sigma^i_{\mathcal{W}}$, without taking un-taintings into account. To *summarize* the predictions of all serializations in $\mathcal{W}$, i.e. to compute $ST_{\mathcal{W}}$, it suffices to take the union of all local predictions. Figure 12 illustrates the enumerative approach.

$$ST_{\mathcal{W}} = \bigcup_{i \in [1,|\mathcal{W}|!]} ST^i_{\mathcal{W}}$$

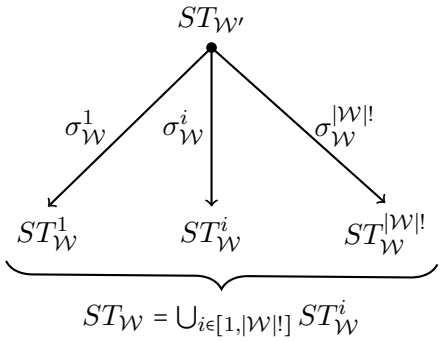


Figure 12: Enumerative prediction of taint propagation

3.4.2 Iterative approach

The enumerative approach presented above has an exponential complexity ($|\mathcal{W}|!$ serializations to process) which makes it impractical. We present here an iterative algorithm with linear complexity for predicting relaxed taintness propagation. Our algorithm iterates over an arbitrary serialization of events in a window at most $|\mathcal{W}|$ times and infers $ST_{\mathcal{W}}$. We justify the correctness of our solution by showing taint prediction is equivalent to solving a *boolean equation system* (BES). Note that the transformations we present hereafter are only used to illustrate the correctness of the solution and never occur in the analysis. In appendix A we provide basic notations and definitions for boolean equation systems.

Equivalence to boolean equation systems

Taintness is a binary value that characterizes a variable as either tainted (*true*, $\top$) or un-tainted (*false*, $\bot$). Thus, taint propagation can be expressed in terms of *boolean equations*. As mentioned earlier, taintness is explicitly propagated to a variable if there exists a tainted variable among those used to define it. By associating to each variable x in the logs a boolean shadow variable denoted as $\widehat{x}$, we can transform an event e_k into an equivalent boolean equation as follows:

$$e_k \equiv \widehat{Def}(e_k) = \bigvee_{x \in Used(e_k)} \widehat{x}$$

Figure 13 illustrates how events in a block can be transformed into an equivalent boolean equation system. The first step transforms each event into an equivalent boolean equation as detailed above. After this first transformation we might have several boolean equations defining the same variable. To obtain a boolean equation system for the block we must eliminate all duplicate definitions. We achieve this by taking the disjunction of all equations defining the same variable. This merging of boolean equations is valid with respect to $taint\mathcal{R}$. Recall that according to $taint\mathcal{R}$ a variable x is tainted if there exists an event that assigns it a tainted value.

The boolean equation system $\mathcal{E}$ we obtain by the above transformation of events consists of *disjunctive* boolean equations. Such boolean equation systems are often represented as directed graphs $G_{\mathcal{E}} = (V, E)$, where $V = \{\ \widehat{x} \mid \widehat{x} \in \mathcal{E}\} \cup \{\top, \bot\}$ is the set of vertices and E is the set of directed edges, representing dependency between variables. Figure 14(a) illustrates the dependency

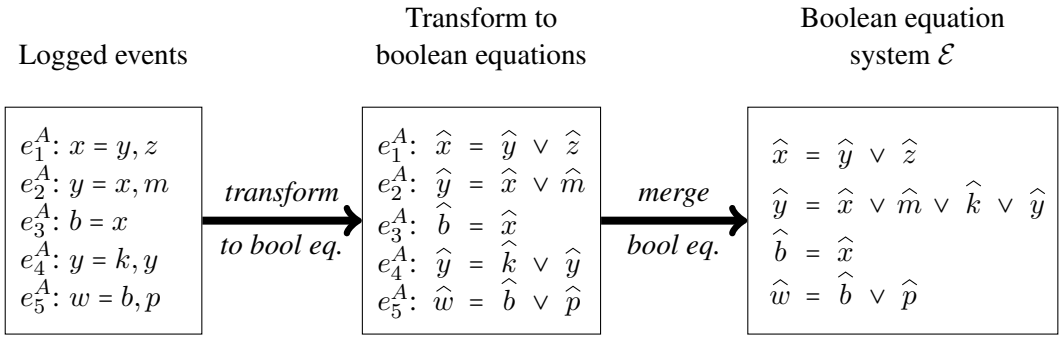
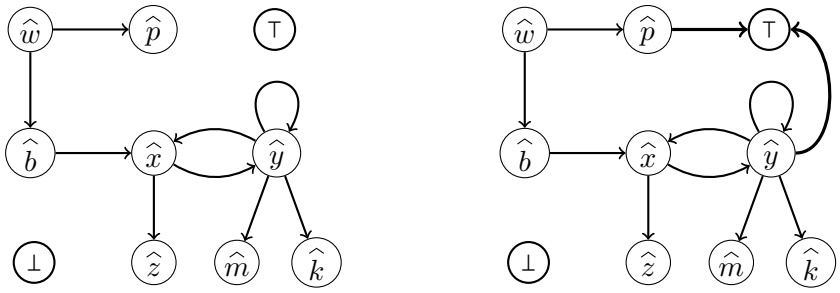


Figure 13: Obtaining boolean equation system.

graph for the BES $\mathcal{E}$ obtained from the block in Figure 13.

$$ST = \{T, y, p\}$$

$$\mathcal{E} \equiv (\ \widehat{x} = \widehat{y} \vee \widehat{z}\ )(\ \widehat{y} = \widehat{x} \vee \widehat{m} \vee \widehat{k} \vee \widehat{y}\ )(\ \widehat{b} = \widehat{x}\ )(\ \widehat{w} = \widehat{b} \vee \widehat{p}\ )$$



(a) Initial BES (b) Updated according to ST

Figure 14: Variable dependency graph of disjunctive boolean equation system.

For each variable $\widehat{x}$ defined in the boolean equation system $\mathcal{E}$ we show that if there exists a solution that makes it *true* then there also exists a serialization σ of logged events such that $taint\mathcal{R}(\sigma, last(\sigma), x)$ holds and conversely.

Finding a solution that assigns a variable $\widehat{x}$ of $\mathcal{E}$ with *true* is equivalent to identifying a path in the dependency graph of the BES that leads from vertex mapped to $\widehat{x}$ to the *true* vertex. Before searching for such a solution we must update the dependency graph such that there exists an edge connecting each variable in the ST with the *true* vertex. This update introduces the information about tainted/*true* variables in the BES. Figure 14(b) illustrates the updated dependency graph with respect to $ST = \{T, y, p\}$. We can now easily identify which variables can reach the *true* vertex. For instance $\widehat{b}$ is assigned a true value through the path ($\widehat{x}$, $\widehat{y}$,$\top$).

We argue now why the existence of a path that propagates *true* value to a variable $\widehat{x}$ implies the existence of a serialization that propagates taintness. The path in the dependency graph defines in which order the equations should be applied such that *true* value reaches the desired equation defining $\widehat{x}$. Under the current assumptions (completely relaxed memory model and not taking untaintings into account) events can be arbitrarily re-ordered. Thus, we can produce a serialization where the ordering of events matches the order imposed on boolean equations. That is, we group all events defining a variable and subsequently order these groups such that they match the ordering of boolean equations. The events defining variables for which the path does not precise an ordering

can be placed arbitrarily.

Figure 15 illustrates a plausible serialization that propagates taintness to b with respect to $taint\mathcal{R}$. On the left side of the figure we have the set of boolean equations that correspond to the block on Figure 13. In the middle we re-order the boolean equations such that *true* value can reach variable $\widehat{b}$. Finally, on the right side we exhibit a serialization that taints b under $taint\mathcal{R}$.

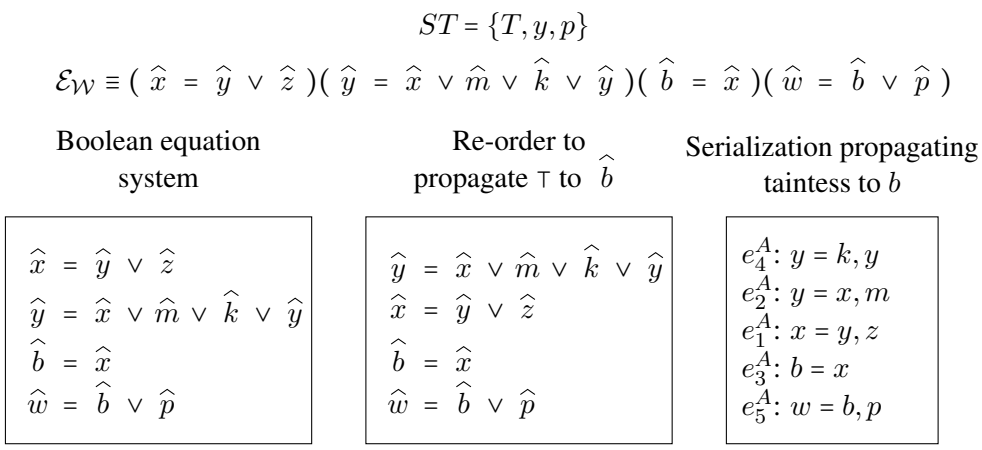


Figure 15: Equivalence between path in dependency graph and tainting serialization

Although there are many efficient algorithms for solving disjunctive BES our taint analysis is based on the iterative one. Note that, as explained in the following section, we do not iterate over the BES itself but directly on the logged events.

Iterative algorithm

As argued above, we can iterate over a block to obtain the predictions of *relaxed taintness*. This can be generalized to a window where we iterate on on an arbitrary serialization $\sigma^{it}_{\mathcal{W}}$. We choose this serialization to be defined as the concatenation of blocks in the window respecting program order. We concatenate first blocks in l_b followed by those in l_t. Figure 16 illustrates the serialization of events that is iterated. The iteration is divided into two phases: (i) *horizontal* and (ii) *vertical*. The *horizontal* phase makes a pass over events in an epoch by crossing blocks left to right. The *vertical* phase iteratively initiates horizontal passes over the *body* and *tail* epochs successively.

Algorithm 2 presents the *vertical* phase, which iterates over the serialization $\sigma^{it}_{\mathcal{W}}$ of events in $\mathcal{W}$. The phases of the algorithm are also illustrated on the left side of Figure 16. The algorithm for horizontal processing of an epoch is provided in Algorithm 3. The horizontal algorithm applies a *transfer* function on each block. Here the transfer function is equivalent to Algorithm 1 where the serialization processed is the blocks events in program order (i.e. as they appear in the log). We must note that the *transfer* function should be monotonic on ST, that is it either adds or removes elements from it. This is required required to terminate the iteration of *vertical* algorithm.

We covered so far the aspect of predicting a relaxed form of explicit taint analysis within a window. We provided an intuitive enumerative method and an equivalent iterative. In subsection 3.5 that follows we abord the sliding window phase of the analysis.

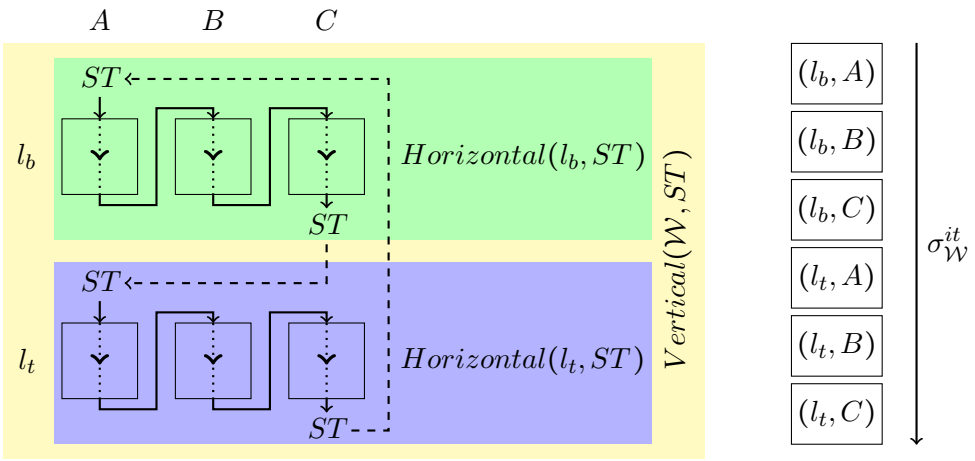


Figure 16: Iterating over the window

Algorithm 2 Vertical processing $Vertical(\mathcal{W}, ST_{\mathcal{W}'})$

In: $\mathcal{W} = \{l_b, l_t\}, ST_{\mathcal{W}'}$
1: $ST \leftarrow ST_{\mathcal{W}'}$
2: **repeat**
3: $\quad ST \leftarrow Horizontal(l_b, ST)$
4: $\quad ST \leftarrow Horizontal(l_t, ST)$
5: **until** (ST **unmodified**)
6: $ST_{\mathcal{W}} \leftarrow ST$
Out: $ST_{\mathcal{W}}$

Algorithm 3 Horizontal processing of epoch $Horizontal(l, ST)$

In: l, ST
1: **for all** block $bl \in l$ **do**
2: $\quad ST \leftarrow Transfer(bl, ST)$
3: **end for**
Out: ST

3.5 Sliding windows - overlapping

As mentioned earlier the slicing of log-files into epochs limits the interleaving of events to be taken into account. Due to the arbitrary slicing we extend the interleaving of events to adjacent epochs. The explicit taint prediction analysis is applied on a sliding window consisting of two epochs. As illustrated in Figure 8 the sliding window allows each event to be interleaved with events in its preceding and succeeding epochs. In window $\mathcal{W}'$ events in epoch l_b are interleaved with events in l_h, while in window $\mathcal{W}$ with events in l_t.

The disjoint processing of interleavings for events in an epoch can affect the approximation of the predictions. That is, they can either over or under approximate explicit taint propagations. These issues are not observable for the *relaxed taintness* where no killing/untainting of variables occurs. We will focus on the effect of sliding window for the case of taint prediction under sequential consistency in section 4.3.

4 Iterative explicit taint propagation under sequential consistency

The iterative prediction is a comfortable and efficient way of predicting taint propagation when any serialization of events is valid and kills are not taken into account. In this section we present how to adapt the iterative algorithm such that *explicit taint* propagation under *sequential consistency* is predicted within a window. Briefly, we must filter out taint propagation that is caused by non sequentially consistent serializations of events. Initially, we maintain the assumption that killing variables is not affecting taint propagation (i.e. they are ignored). We remind that *sequential consistency* enforces (i) program order and (ii) write atomicity. Write atomicity is meaningful only for parallel executions, thus it does not affect reasoning on serializations as is the case.

4.1 Respecting program order without kills

The taint property we are interested in is *relaxed taintness* propagation (see Definition 3.3) applied to *sequentially consistent serializations*. We remind the precedence binary operator $\blacktriangleleft$ which defines ordering of events for a single thread (order in which events of a thread were logged). Furthermore, we introduce a more general binary operator $\lhd$ which denotes precedence between events produced by any thread and respecting the window interleaving assumption. We remind the more detailed notation of events for a given slicing $e_i^t \equiv (l,t,i)$, where $Epoch(e_i^t) = l$. The operators are formalized as follows:

$$(l,t,i) \lhd (l',t',j) \equiv \big(t = t' \wedge \big(l < l' \vee (l = l' \wedge i < j)\big)\big) \vee \big(t \neq t' \wedge l' \geq l - 1 \big)$$

$$(l,t,i) \blacktriangleleft (l',t',j) \equiv t = t' \text{ and } (l,t,i) \lhd (l',t',j)$$

We provide here the definition of a sequentially consistent serialization consisting of events belonging to a window $\mathcal{W}$. Since events are restricted to a window, the events belonging to different threads can appear in any order. Though, for an event e_m in the serialization we must ensure that all events e_k in the same thread that precede it ($e_k \blacktriangleleft e_m$) also precede it in the serialization.

$$\sigma^i_{\mathcal{W}} = \{(e_1,...,e_n) \mid \forall\, k,m \in [1,n] \text{ s.t. } k < m \Rightarrow e_k \lhd e_m \\ \forall\, m \in [1,n]\,,\, e_k \in \mathcal{W} \text{ s.t. } e_k \blacktriangleleft e_m \Rightarrow e_k \in \sigma^i_{\mathcal{W}} \wedge k < m\}$$

We provide hereafter an example on which we apply the iterative taint prediction as presented earlier and sketch the proposed adaptation. Figure 17 illustrates a window consisting of two blocks (l_b, A) and (l_b, B) (the tail epoch is empty), which are iterated in order A, B. On the right side are the summaries obtained by each iteration. We focus on the second iteration where variables `y,w,d` are marked as tainted. While the tainting of variables `y,w` respects program order, that of variable `d` does not. The serialization $\sigma^i_{\mathcal{W}}$ for which $taint\mathcal{R}(\sigma^i_{\mathcal{W}}, \mathtt{d}, e_1^B)$ holds is the following $(e_2^B, e_1^A, e_2^A, e_3^A, e_1^B)$. Executing e_2^B before e_1^B does not conform with *program order* and thus tainting of `d` should not be included in the taint predictions.

To filter out taint predictions that do not respect program order, we should verify that for each predicted taint propagation there exists a sequentially consistent serialization of events that justifies it (note that, still killings/untaintings of variables are ignored). To do so, we keep track of

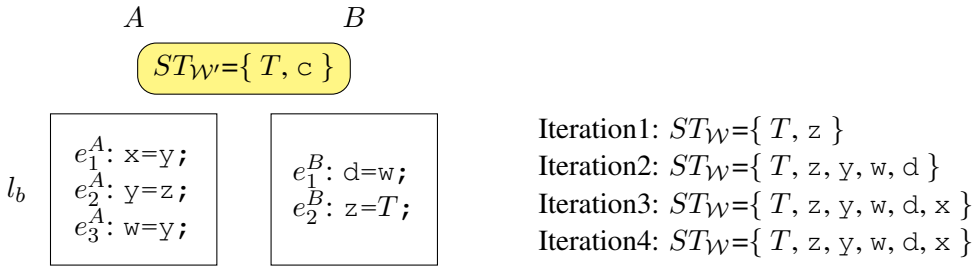


Figure 17: Example, sequential consistency and iteration

events that taint variables as well as the tainting source (i.e. through which variables taintness is propagated). This information about generated/tainted variables is stored in what we call the *gen history* of the window ($GH_{\mathcal{W}}$). $GH_{\mathcal{W}}$ maps each tainted variable x to a set of *markings*. A *marking* $\mathfrak{m}$ is a pair (e, V) where e is the event that taints x ($Def(e) = x$) and V is the set of variables that cause it to be tainted ($V \subseteq Used(e)$). The *gen history* stores information for the currently analyzed window only. We define the function $GH_{\mathcal{W}}(x)$ which returns the markings associated to a variable x inside a window $\mathcal{W}$.

$$GH_{\mathcal{W}}(x) = \{\, (e,V) \mid Def(e) = x \;\bigwedge\; \forall y \in V:\; y \in Used(e) \;\wedge\; \exists\, \sigma^i_{\mathcal{W}} \text{ such that } taint\mathcal{R}(\sigma^i_{\mathcal{W}}, e, y) \,\}$$

Vars	$ST_{\mathcal{W}'}$	Iteration1	Iteration2	Iteration3
T	✓			
c	✓			
z		$(e_2^B,\{T\})$		
y			$(e_2^A,\{\text{z}\})$	
w			$(e_3^A,\{\text{y}\})$	
d			$(e_1^B,\{\text{w}\})$	
x				$(e_1^A,\{\text{w}\})$

Table 1: Gen history example

Table 1 illustrates the usage of $GH_{\mathcal{W}}$ for the example of Figure 17. During first iteration we update $GH_{\mathcal{W}}(\text{z})$ with the marking $(e_2^B, \{T\})$. The marking encodes the information that variable z was tainted at event e_2^B, and that the variable that caused it to be tainted is T which belongs to $ST_{\mathcal{W}'}$. Similarly, each successive iteration adds the markings accordingly. $GH_{\mathcal{W}}$ captures all the necessary information to track the source of tainting inside the window, and infer whether it respects sequential consistency or not. We point out with a colored background the invalid markings, i.e. false taint propagations. The filtering of false predictions can occur either online, the variable is neither added to the ST nor a marking is added to $GH_{\mathcal{W}}$, or offline. Finally,

we introduce the function $Events(GH_{\mathcal{W}}(x))$ which returns a set containing all events that taint variable x.

Definition 4.1 ($Events(GH_{\mathcal{W}}(x))$ **)**

Returns the set containing all events that taint variable x according to $GH_{\mathcal{W}}$.

$$Events(GH_{\mathcal{W}}(x)) = \{e_k \mid \exists\, \mathfrak{m} = (e_k, V) \in GH_{\mathcal{W}}(x)\}$$

We provide in Algorithm 4 the *transfer function* to be applied on blocks such that $GH_{\mathcal{W}}$ is updated properly and used to filter out online the non sequentially consistent taint propagations. Of utmost importance is function $TaintingVars$ which returns the set of variables that can taint the variable defined by the current event e. If the set of tainting variables tv is not empty, then we update the $GH_{\mathcal{W}}$ accordingly by adding the mapping $\big(Def(e), (e, tv)\big)$ (line 4) and also update the set of taint predictions (at line 5).

Algorithm 4 Transfer function for taint analysis

In: B , ST
1: **for all** $e \in B$ **do**
2: $tv = TaintingVars(e)$;
3: **if** $tv \neq \varnothing$ **then**
4: $GH_{\mathcal{W}} \leftarrow GH_{\mathcal{W}} \cup \{\big(Def(e)\,,\,(e, tv)\big)\}$;
5: $ST \leftarrow ST \cup Def(e)$;
6: **else**
7: // ignore kills, do nothing
8: **end if**
9: **end for**
Out: ST

To define function $TaintingVars$ we need to introduce first the notion of *taint dependency path*. We start with a more generic definition, that of a backward dependency path $\mathcal{P}^b$, which is a sequence of events that form a chain of defined and used variables.

Definition 4.2 (Backward dependency path $\mathcal{P}^b$)

$$\mathcal{P}^b = \{(e_1, \ldots, e_n) \mid \forall k \in [1, n-1]\,.\, Used(e_k) \cap Def(e_{k+1}) \neq \varnothing\}$$

A *taint dependency path* $\mathcal{P}$ for a variable x is a backward dependency path limited to a window $\mathcal{W}$. The path is retrieved in $GH_{\mathcal{W}}$ and ends in the initial set of tainted variables $ST_{\mathcal{W}'}$ where we recall $\mathcal{W}'$ is the window preceding $\mathcal{W}$.

Definition 4.3 (Taint dependency path $\mathcal{P}$)

$$\begin{aligned}\mathcal{P} = \{(e_1, \ldots, e_n) \mid\ & \forall k \in [1, n] : e_k \in \mathcal{W} \ \wedge \\ & \forall k \in [1, n-1] : Used(e_k) \cap Def(e_{k+1}) \neq \varnothing \ \wedge \\ & \qquad Def(e_1) = x \ \wedge\ Used(e_n) \cap ST_{\mathcal{W}'} \neq \varnothing \ \wedge \\ & \qquad \exists\, y, e_m, V \textit{ such that } (e_m, V) \in GH_{\mathcal{W}}(y) \ \wedge\ e_m = e_{k+1}\ \}\end{aligned}$$

A taint dependency path is the sequence of events that correspond to the recursive invocations of $taint\mathcal{R}$. Recalling Figure 11 on page 21 the taint dependency path for variable x is $\mathcal{P} = (e_m, e_v)$. Note that, inversing a taint dependency path $\mathcal{P}$ produces a partial serialization, enforcing the ordering of some key events, such that $Def(e_1)$ gets tainted, where e_1 is the first event in $\mathcal{P}$. We use the notation $\sigma(\mathcal{P})$ to represent the partial serialization of events corresponding to a taint dependency path $\mathcal{P}$. That is, given $\mathcal{P} = (e_1, e_2, e_3)$ then $\sigma(\mathcal{P}) = (e_3, e_2, e_1)$. Moreover, we call $TDP(GH_{\mathcal{W}}, x)$ the set of taint dependency paths for variable x with respect to $GH_{\mathcal{W}}$. The set of paths can be obtained with a recursive exploration of markings for variable x.

Back to the definition of $TaintingVars(e)$ in Algorithm 4. The function computes the set tv of variables that taint $Def(e)$ (the variable defined by e). For every variable $y \in tv$ there must exist a taint dependency path $\mathcal{P}$ such that: if e is added to it as the first event, the resulting path is *valid*. The definition of a *valid path* can be modified accordingly to capture any restrictions that must apply to serializations of events (i.e. capture different weak memory models). For now a path is valid if the events respect sequential consistency. For sequential consistency, the call to $isValid(\mathcal{P})$ is equivalent to calling $isConsistent(\mathcal{P})$ which we define here:

Definition 4.4 (Predicate $isConsistent(\mathcal{P})$)

A taint dependency path $\mathcal{P}$ is sequentially consistent if the serialization of events it defines, which is the inverse order of events, is sequentially consistent. Thus predicate $isConsistent(\mathcal{P})$ is defined as:

$$\forall e_k, e_m \in \mathcal{P} \text{ then } (k < m \Rightarrow e_m \lhd e_k)$$

Algorithm 5 presents the computation of tv. For each variable in $Used(e)$ it obtains its taint dependency paths and extends them by adding event e as the first event (line 12). We use the following notation $e.\mathcal{P}$ to denote the concatenation of paths or of an event e with a path $\mathcal{P}$. The resulting path is checked for validity. If it is valid then variable $y \in Used(e)$ can successfully taint $Def(e)$ and is added to tv. There is a special case where variable y belongs to $ST_{\mathcal{W}'}$. In this case we produce an immediate path consisting uniquely of event e and after checking it for validity we add y to tv.

To clarify the definition of $TaintingVars(e)$ we apply it to the example of Figure 17 and the corresponding illustration of its *gen history* in table 1. First, we apply $TaintingVars(e_2^B)$ during first iteration. Since $Used(e) = T$ which belongs to $ST_{\mathcal{W}'}$ the path to check is $\mathcal{P} = (e_2^B)$ which respects sequential consistency. Thus, variable T is added to tv. We apply now $TaintingVars(e_1^B)$ during the second iteration. There is only one taint dependency path for variable `w` which is $\mathcal{P} = (e_3^A, e_2^A, e_2^B)$. We add e_1^B as the first element, $\mathcal{P} = e_1^B.\mathcal{P} = (e_1^B, e_3^A, e_2^A, e_2^B)$. The resulting path $\mathcal{P}$ is not sequentially consistent because e_2^B does not precede e_1^B ($e_2^B \not\lhd e_1^B$).

Note that in this section we make the assumption that kills are ignored. Thus, if there exists a sequentially consistent path $\mathcal{P}$ that generates a variable `x`, then there also exists a serialization $\sigma^i_{\mathcal{W}}$ that generates it. To produce $\sigma^i_{\mathcal{W}}$ it suffices to extend the partial serialization $\sigma(\mathcal{P})$ obtained from $\mathcal{P}$ such that all remaining events in $\mathcal{W}$ are positioned on $\sigma^i_{\mathcal{W}}$ respecting program order for all threads. Based on the assumption the effect of these events is ignored.

Algorithm 5 $TaintingVars(e)$, set of variables that produce valid taint propagation

In: e

1: $tv \leftarrow \varnothing$
2: **for all** $y \in Used(e)$ **do**
3: **if** $y \in ST_{\mathcal{W}'}$ **then**
4: $\mathcal{P} \leftarrow e$ // $\mathcal{P}$ is a path consisting of just event e
5: **if** $isValid(\mathcal{P})$ **then**
6: $tv \leftarrow y \cup tv$
7: continue;
8: **end if**
9: **end if**
10: $TP_y \leftarrow TDP(GH_{\mathcal{W}}, y)$
11: **for all** $\mathcal{P} \in TP_y$ **do**
12: $\mathcal{P} \leftarrow e \,.\, \mathcal{P}$ // add event e as first event of $\mathcal{P}$
13: **if** $isValid(\mathcal{P})$ **then**
14: $tv \leftarrow y \cup tv$
15: break;
16: **end if**
17: **end for**
18: **end for**

Out: tv

We illustrate in Figure 18 the composition of a serialization. On the left side we illustrate the currently analyzed window (for clarity we assume its tail epoch is empty) consisting of two blocks. The arrows depict the taint dependency path $\mathcal{P}$ that taints x. On the right side, we explicit the path $\mathcal{P}$ and the derived partial serialization $\sigma(\mathcal{P})$. Below it we appose the remaining events denoted as $\mathcal{W} \setminus \mathcal{P}$. An arrow initiated from each remaining event indicates its plausible positioning such that a sequentially consistent $\sigma^i_{\mathcal{W}}$ is obtained.

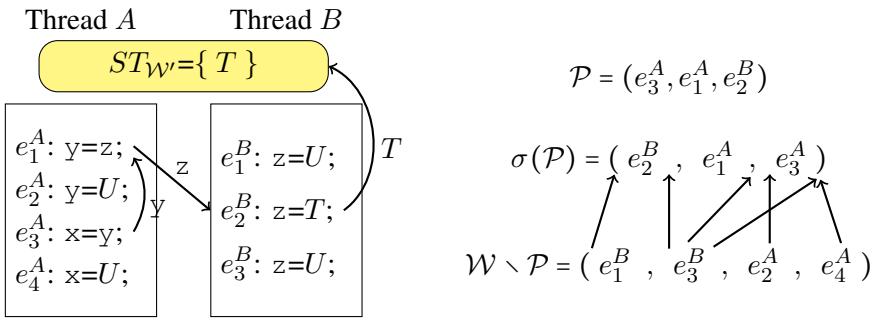


Figure 18: Composing a sequentially consistent serialization based on a TDP

4.2 Taking kills into account

In this section we introduce the killing/un-tainting of variables. Taking kills into account makes taint predictions more accurate and thus reduces *false positives*. Though, extra care must be taken since we do not want our analysis to miss any valid taint propagations. The killing of a variable affects taint prediction in two ways:

i) it prevents taint propagation between variables;

ii) it excludes variables from the summarization of the window.

Prior to detailing the two cases we give their intuition using the example of Figure 18. In the first case, kill/untainting of a variable occurs between the tainting point of a variable and its usage to propagate taintness to another variable. In the example event e_2^A must be executed between e_1^A and e_3^A. With kills taken into account the given path $\mathcal{P}$ cannot produce a serialization such that x is tainted. For the second case we shall focus on variable z which is explicitly tainted at event e_2^B. Note that the succeeding event e_3^B untaints z. Thus on all sequentially consistent serializations of the window variable z will eventually be untainted and thus should not be included in the taint predictions of the window (i.e. $ST_{\mathcal{W}}$).

We approach the killing/untainting of variables by separating the two cases identified. The killing of variables that break tainting paths is treated *online* during the iterative algorithm. Dually, the killing of variables that excludes them from the summarization of the window is treated *offline* i.e. after the iterative algorithm (Algorithm 2)has completed.

4.2.1 Killing a taint dependency path (TDP)

For the killing of tainting paths $\mathcal{P}$ we need to verify that there exists a sequentially consistent partial serialization $\sigma^i_{\mathcal{P}}$ consisting of all events preceding the events in $\mathcal{P}$, such that $taint\mathcal{W}(\sigma^i_{\mathcal{P}}, e_1, Def(e_1))$ (see Definition 3.2) holds, where e_1 is the first event in $\mathcal{P}$. Figure 19 illustrates with a light background the events that must be included in $\sigma^i_{\mathcal{P}}$. In this abstract example the path $\mathcal{P}$ is designated by the arrows. The check for the existence of a $\sigma^i_{\mathcal{P}}$ is performed online during the computation of $TaintingVars$ (see Algorithm 5) as part of the $isValid(\mathcal{P})$ predicate. More precisely, predicate $isValid(\mathcal{P})$ is the conjunction of predicates $isConsistent(\mathcal{P})$ (see Definition 4.4) and $noKill(\mathcal{P})$ which we define after the presentation of some key examples.

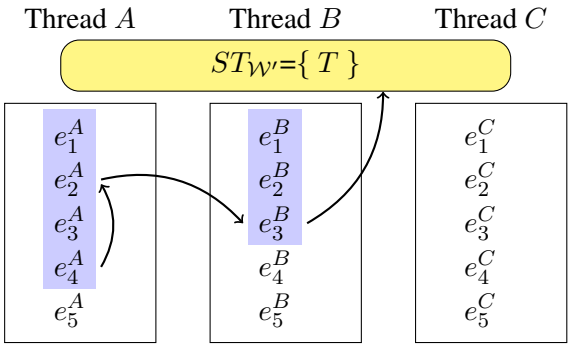
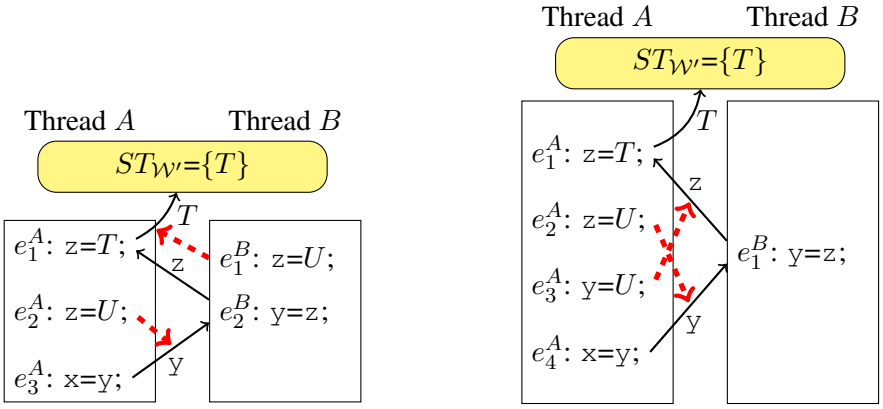


Figure 19: Events contained in $\sigma^i_{\mathcal{P}}$

Figure 20 illustrates two examples. In both examples the solid arrows, labeled by the variable that propagates taintness, represent the tainting path $\mathcal{P}$ that causes variable x to be tainted at events e_3^A and e_4^A accordingly. The dashed thick arrows designate where the killing events should be placed such that they do not break $\mathcal{P}$. In Figure 20(a) event e_2^A must be positioned after e_2^B such that it kills variable z only after it has propagated its taintness to variable y. Similarly e_1^B must be placed before e_1^A such that it kills z before it gets tainted. In this example the killing events can be

serialized such that $\mathcal{P}$ is capable of tainting x. Dually, in Figure 20(b) the path $\mathcal{P}$ does not hold. As illustrated e_2^A must be place after e_1^B while inversely e_3^A before e_1^B. Due to the program order imposed one of the two events will inevitably break $\mathcal{P}$.



(a) valid serialization exists (b) no valid serialization exists

Figure 20: Inferring a valid serialization for a path $\mathcal{P}$

To verify if there exists a serialization $\sigma_{\mathcal{P}}^i$ such that the killing events do not break $\mathcal{P}$ we perform some sanity checks on the composition of $\mathcal{P}$ with the preceding events. Here are the observations that allow us to make these checks in a incremental way.

- all events that are not part of $\mathcal{P}$ are considered as kills. Figure 21 illustrates an example where two tainting paths are present, one defined by solid and the other by dashed arrows. The solid path is considered as invalid because event e_2^A is considered a kill of variable z witch breaks the solid path. Variable x though gets tainted by the dashed path.

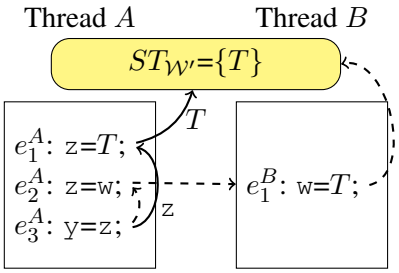


Figure 21: Events not belonging to $\mathcal{P}$ are considered kills

- events belonging to different threads can be serialized independently

Before giving the definition of predicate $noKill(\mathcal{P})$ we introduce the following notations:

$\sigma^a \oplus \sigma^b$ is a sequentially consistent merge operator. Given two serializations of events σ^a and σ^b, themselves respecting sequential consistency, it produces all plausible sequentially consistent serializations containing events of σ^a and σ^b.

$\sigma(e_k^A, e_m^A)$ defines a sub-sequence of events produced by a thread. The events contained are scoped by e_k^A and e_m^A, where $e_k^A \blacktriangleleft e_m^A$. For example, $\sigma(e_1^A, e_6^A) = (e_2^A, e_3^A, e_4^A, e_5^A)$

$taintSource(\mathcal{P})$ returns the tainting source, i.e. the variable that is reached by $\mathcal{P}$ in $ST_{\mathcal{W}'}$. In the example of Figure 21 $taintSource(\mathcal{P}) = T$ (both for the solid and dashed paths).

Definition 4.5 (Predicate $noKill(\mathcal{P})$)

Under sequential consistency, a taint dependency path $\mathcal{P}$ is not broken by a killed variable if there exists a sequentially consistent serialization of events belonging in $\mathcal{P}$ and the events preceding them such that, between two successive events e_i, e_{i+1} of $\mathcal{P}$ there is no event e_j such that $Def(e_j) = Def(e_i)$.

$$\forall e_k \in \mathcal{P} = (e_1, ..., e_k, ..., e_m, ..., e_n) \text{ we distinct the following cases:}$$

- $\exists\, m > k$ *such that* $e_m \blacktriangleleft e_k \ \wedge\ \nexists e_q$ *such that* $m > q > k \ \wedge\ e_m \blacktriangleleft e_q$
  - *if* $m = k + 1$ *then* (e.g. Figure 21 e_3^A,e_1^A)

$$\forall e_j \text{ s.t. } e_m \blacktriangleleft e_j \blacktriangleleft e_k \ \Rightarrow\ Def(e_j) \neq Def(e_m)$$

  - *else* (e.g. Figure 20(b) e_4^A,...,e_1^A)

$$\exists\, \sigma = (\ldots, e_i, \ldots, e_j, \ldots, e_{i+1}, \ldots) \in \sigma(\mathcal{P}) \oplus \sigma(e_m, e_k) \text{ s.t.}$$
$$\forall e_i, e_{i+1} \in \sigma(\mathcal{P}), e_j \in \sigma(e_m, e_k) \ \Rightarrow\ Def(e_j) \neq Def(e_i)$$

- $\nexists\, m > k$ *such that* $e_m \blacktriangleleft e_k$
  - *if* $k = n$ *then* (e.g. Figure 20(b) e_1^A)

$$\forall e_j \text{ such that } e_j \blacktriangleleft e_k \ \Rightarrow\ Def(e_j) \neq taintSource(\mathcal{P})$$

  - *else (we introduce a dummy event)* (e.g. Figure 20(b) e_2^B)

$$\mathcal{P}' = \mathcal{P}.e_{k'} \text{ where } Def(e_{k'}) = taintSource(\mathcal{P}) \wedge$$
$$\forall e_j \in \mathcal{W} \text{ such that } Thr(e_j) = Thr(e_k) \ \Rightarrow\ e_{k'} \blacktriangleleft e_j$$

  Now we can apply the first case where:

$$\exists\, m > k \text{ such that } e_m \blacktriangleleft e_k \ \wedge\ \nexists e_q \text{ such that } e_m \blacktriangleleft e_q$$

(note that $m > k + 1$)

To simplify the definition of predicate $noKill(\mathcal{P})$ in the case were $m > k + 1$ we make use of the merge operator $\oplus$ implying that all sequentially consistent serializations consisting of events in $\sigma(\mathcal{P})$ and $\sigma(e_m, e_k)$ are computed. This may be misleading since we stated earlier that we verify the existence of $\sigma^i_{\mathcal{P}}$ incrementally. We provide here the checks that verify that:

$$\exists\, \sigma = (\ldots, e_i, \ldots, e_j, \ldots, e_{i+1}, \ldots) \in \sigma(\mathcal{P}) \oplus \sigma(e_m, e_k) \text{ s.t.}$$
$$\forall e_i, e_{i+1} \in \sigma(\mathcal{P}), e_j \in \sigma(e_m, e_k) \ \Rightarrow\ Def(e_j) \neq Def(e_i)$$

Definition 4.6 (Positioning kill events)

Given a taint dependency path $\mathcal{P} = (e_1, \ldots, e_k, \ldots, e_m, \ldots, e_n)$ *where* $e_m \blacktriangleleft e_k$ *and* $m > k+1$. *We want to check that events in* $\sigma(e_m, e_k)$ *can be ordered such that they respect sequential consistency and do not kill a variable between the point it is defined and used to propagate taintness. That is between events* e_i, e_{i+1} *in* $\sigma(\mathcal{P})$ *where* $k \leq i < m$. *To respect program order, for an event* e_j $(e_m \blacktriangleleft e_j \blacktriangleleft e_k)$ *that kills a variable of* $\sigma(\mathcal{P})$ *all events preceding should be able to be positioned before it and dually all succeeding events after it.*

$$\textit{Given } \mathcal{P} = (e_1, \ldots, e_k, \ldots, e_m, \ldots, e_n) \textit{ where } m > k+1 \textit{ and } e_m \blacktriangleleft e_k\textit{:}$$

- $\forall e_j$ *s.t.* $e_m \blacktriangleleft e_j \blacktriangleleft e_k \;\Rightarrow\; Def(e_j) \cap \left(\bigcup_{i \in [k+1,m]} Def(e_i)\right) \neq \varnothing$

$$\bigvee$$

- $\forall e_j\,,\ i \in [k+1, m]$ *s.t.* $e_m \blacktriangleleft e_j \blacktriangleleft e_k \;\wedge\; Def(e_j) = Def(e_i)$

$$\textit{then:} \begin{cases} \forall e_{j'} \textit{ s.t. } e_m \blacktriangleleft e_{j'} \blacktriangleleft e_j \;\Rightarrow\; \exists i' \in [i+1, m] \textit{ s.t. } Def(e_{i'}) \neq Def(e_{j'}) \\ \bigvee \\ \forall e_{j'} \textit{ s.t. } e_j \blacktriangleleft e_{j'} \blacktriangleleft e_k \;\Rightarrow\; \exists i' \in [k+1, i-1] \textit{ s.t. } Def(e_{i'}) \neq Def(e_{j'}) \end{cases}$$

To conclude taking into account kills in taint propagation we remind that the predicate $noKill(\mathcal{P})$ is used in conjunction with $isConsistent(\mathcal{P})$ in the call of $isValid(\mathcal{P})$ at Algorithm 5. Thus, the gen history of the current window ($GH_{\mathcal{W}}$) is precisely updated.

4.2.2 Excluding variables from window summarization

As mentioned earlier the killing of variables such that they are excluded from the summarization of a window are treated offline. After the iterations of Algorithm 2 have completed we hold $ST_{\mathcal{W}}$ which over-approximates the set of tainted variables down to $\mathcal{W}$ and $GH_{\mathcal{W}}$ which contains the markings for generated variables. For each marking $\mathfrak{m}$ in $GH_{\mathcal{W}}$ the following is true:

$$\mathfrak{m} = (e_k, V) \;\in\; GH_{\mathcal{W}} \;\Leftrightarrow\; \exists\, \sigma^i_{\mathcal{P}} \text{ s.t. } taintW(\sigma^i_{\mathcal{P}}, e_k, Def(e_k))$$

We remind that, a variable x is included in the summarization of the window if there exists a serialization $\sigma^i_{\mathcal{W}}$ of all events in $\mathcal{W}$ such that $taintW(\sigma^i_{\mathcal{W}}, last(\sigma^i_{\mathcal{W}}), x)$ holds. Dually, to exclude a variable x from the summarization then on all serializations $\sigma^i_{\mathcal{W}}$ the last assignment to x should be with an untainted value. Since we do not construct all serializations, but instead use the iterative algorithm which guarantees us to identify all propagations, we cannot precisely (at least not cost-effectively) identify variables killed in $\mathcal{W}$. Thus, we under-approximate the killing of variables by identify the following cases for which we are certain variables are killed in $\mathcal{W}$:

- x is defined and never generated

$$\exists e_k \in \mathcal{W} \text{ s.t. } Def(e_k) = x \;\wedge\; GH_{\mathcal{W}}(x) = \varnothing$$

- all threads that generate x successively kill it

$$\forall e_k \in Events(GH_{\mathcal{W}}(x)) \Rightarrow \exists e_m \in \mathcal{W} \text{ s.t. } e_k \blacktriangleleft e_m \wedge Def(e_m) = x \wedge e_m \notin Events(GH_{\mathcal{W}}(x))$$

The first case is straight forward. Variable x is defined in $\mathcal{W}$ but no marking exists for it in $GH_{\mathcal{W}}$. Thus, all events in $\mathcal{W}$ that define it assign an untainted value. We conclude that on all serializations $\sigma^i_{\mathcal{W}}$ x is killed and thus can be removed from $ST_{\mathcal{W}}$. Figure 22 illustrates an abstract example where only the code of events that define x is given explicitly. We assume that there does not exist any valid tainting path such that variable d is tainted at e^C_1. Thus, on all serializations $\sigma^i_{\mathcal{W}}$ variable x is lastly assigned an un-tainted value. As illustrated x is removed from $ST_{\mathcal{W}}$.

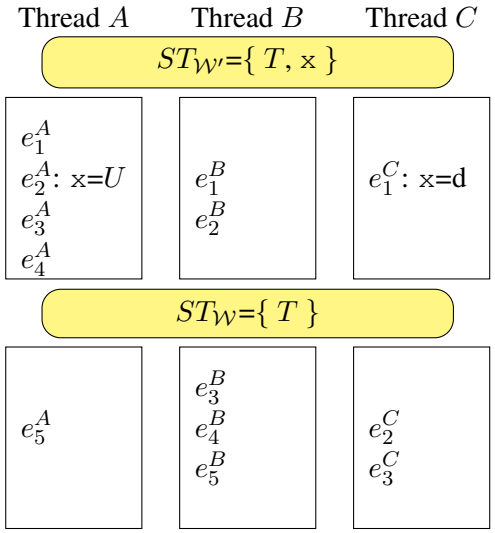


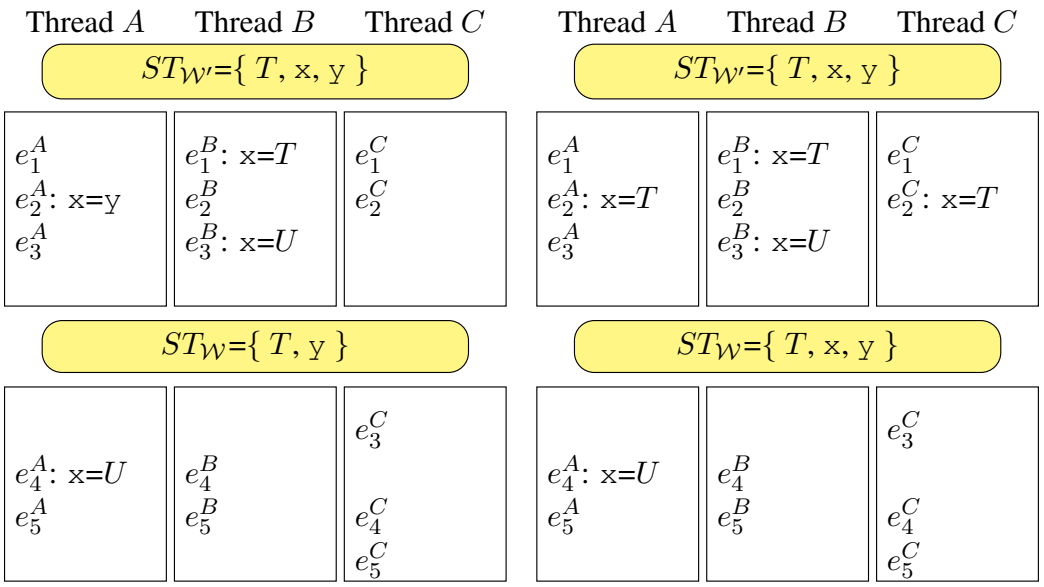
Figure 22: Kill in $\mathcal{W}$ when not generated

In the second case variable x is generated. Thus, there exists at least a partial serialization $\sigma^i_{\mathcal{P}}$ for which x is tainted at event e_k. To ensure that finally x is assigned an untainted value, on all plausible serializations, it must be killed by an event e_m of the same thread that succeeds e_k i.e. $e_k \blacktriangleleft e_m$. Figure 23(a) illustrates an abstract example where all threads that taint x successively kill it. We can note that program order guarantees that eventually the last assignement to x is always an untainted value. Hence, x is safely excluded from $ST_{\mathcal{W}}$. Dually in Figure 23(b) thread C generates x at event e^C_1 and there does not exist any succeeding event in C that kills x. Obviously for the serialization $\sigma^i_{\mathcal{W}} = (\ldots, e^C_2, e^C_3, e^C_4, e^C_5)$ x ens up tainted.

To conclude removing killed variables from the summarization we provide Algorithm 6 which updates Algorithm 2 by adding the offline processing that refines the summarization of the currently analyzed window. We also give in an algorithmic-like form the removing of killed variables in Algorithm 7.

4.3 Effects of sliding window

We mentioned earlier in section 3.3 that there are two aspects in our sliding window-based analysis: (i) predicting explicit taint propagations within a window and (ii) reasoning correctly about



(a) Kill after generating (b) Generating without killing

Figure 23: All threads generating x must eventually kill it

Algorithm 6 Vertical processing $Vertical(\mathcal{W}, ST_{\mathcal{W}'})$

In: $\mathcal{W} = \{l_b, l_t\}, ST_{\mathcal{W}'}$
1: $ST \leftarrow ST_{\mathcal{W}'}$
2: **repeat**
3: $ST \leftarrow Horizontal(l_b, ST)$
4: $ST \leftarrow Horizontal(l_t, ST)$
5: **until** (ST **unmodified**)
6: $ST_{\mathcal{W}} \leftarrow WindowKills(ST, GH_{\mathcal{W}})$
Out: $ST_{\mathcal{W}}$

the overlapping of interleavings caused by sliding windows.We develop here how the sliding of windows affects our predictions, and what precautions can be taken such that our predictions remain an over-approximation of taintness, but also increase the confidence on the soundness of the predictions.

4.3.1 Killing variables in tail causes under-approximates taint propagation

We focus on the killing of variables within a window. The definition we gave before is correct when restricted to a window. When sliding windows, it may cause *under-approximation* of taint predictions. Figure 24 illustrates such an example. We note that, in window $\mathcal{W}'$ variable x is killed since on all serializations $\sigma^i_{\mathcal{W}'}$ it is finally assigned a non-tainted value. Though, the killing is premature because it eliminates the propagation of taintness to variable z. The propagation is feasible under our assumptions since e_1^B can be executed prior the untainting of x. The arrow shows the interleaving under which taintness is propagated in $\mathcal{W}$.

The example of Figure 24 shows that kills that occur in the tail epoch of a window may hide valid taint propagations on the consecutive window. To overcome these issues, we shall only

Algorithm 7 $WindowKills(ST, GH_{\mathcal{W}})$

In: $ST, GH_{\mathcal{W}}$

$ST_{\mathcal{W}} \leftarrow ST$
for all $x \in ST$ **do**
  if $GH_{\mathcal{W}}(x) = \varnothing \ \wedge \ \exists e_k \in \mathcal{W}$ s.t. $Def(e_k) = x$ **then**
    $ST_{\mathcal{W}} \leftarrow ST_{\mathcal{W}} \setminus \{x\}$
  else
    no_kill_exists $\leftarrow false$
    for all $e_k \in Events(GH_{\mathcal{W}}(x))$ **do**
      if $\nexists e_m \in \mathcal{W}$ s.t. $Def(e_m) = x \ \wedge \ e_k \blacktriangleleft e_m \ \wedge \ e_m \notin Events(GH_{\mathcal{W}}(x))$ **then**
        no_kill_exists $\leftarrow true$
        break;
      end if
    end for
    if no_kill_exists = $false$ **then**
      $ST_{\mathcal{W}} \leftarrow ST_{\mathcal{W}} \setminus \{x\}$
    end if
  end if
end for

Out: $ST_{\mathcal{W}}$

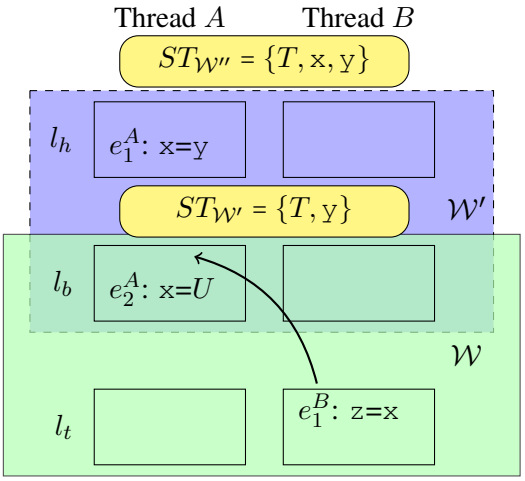


Figure 24: Delay killing in tail

exclude a variable from the summarization if it is killed in the body of a window.

4.3.2 Incompatible TDPs over-approximate taint propagation

Windows consisting of two epochs allow us to reason only on valid interleavings, based on our initial assumptions, but has the disadvantage of predicting possibly incompatible propagations. For two consecutive windows $\mathcal{W}'$, $\mathcal{W}$ the interleavings of events in the common epoch l_b with $l_h$in $\mathcal{W}'$ and l_t in $\mathcal{W}$ are computed independently. Thus, it is possible that the serializations that propagate taintness in $\mathcal{W}'$ and $\mathcal{W}$ are conflicting.

Figure 25 illustrates an example of incompatible TDPs. In the first window $\mathcal{W}' = \{l_h, l_b\}$

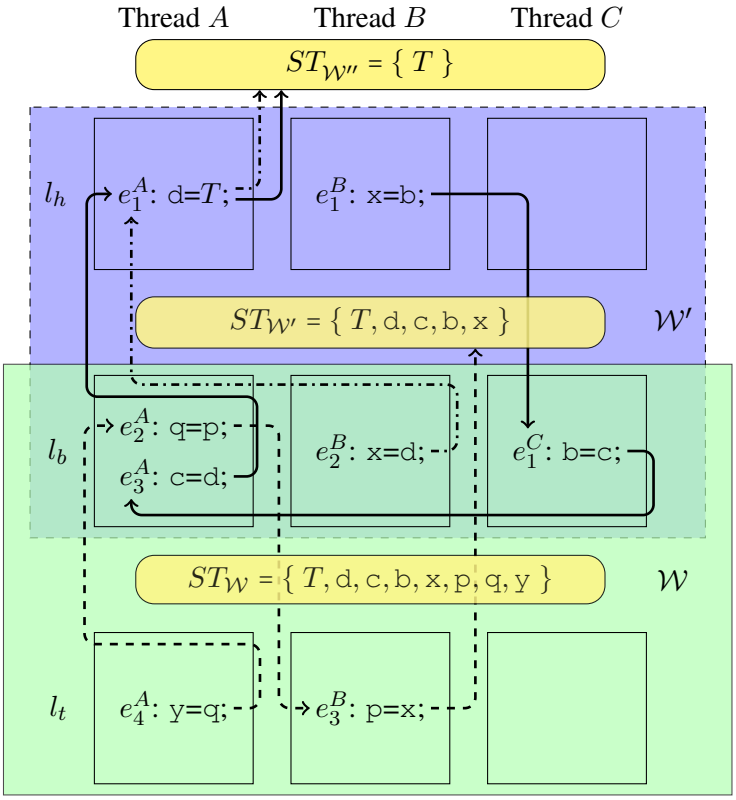


Figure 25: Propagation through incompatible TDPs

variable x is tainted through $\mathcal{P}_1 = (e_1^B, e_1^C, e_3^A, e_1^A)$ traced with a solid line. The summary $ST_{\mathcal{W}'}$ correctly contains x as there exists a serialization which taints x. Sliding to the next window $\mathcal{W} = \{l_b, l_t\}$ we identify with a dashed line $\mathcal{P}_2 = (e_4^A, e_2^A, e_3^B)$ which taints variable y using x. Taint dependency path $\mathcal{P}_2$, although valid in $\mathcal{W}$, is *not compatible* with the one that generated x (which is the tainting source for variable y). Namely, the two TDPs cannot be merged and hence they do not provide a concrete serialization demonstrating how y gets tainted.

Merging TDPs computed in different windows is not always feasible. However, TDPs indicating why a variable is tainted within a window are not unique (although finding a single path is sufficient in our algorithm). For instance, a closer look at Figure 25 shows a second $\mathcal{P}_3 = (e_2^B, e_1^A)$ (with dash-dotted path) for variable x which is compatible with $\mathcal{P}_2$.

The incompatibility of TDPs over-approximates our predictions. To reduce the number of false positives we can classify the tainted variables into two categories *strong* and *weak*. *Strongly* tainted variables are those for which taintness propagation occurred through mergeable tainting paths. Dually, *weakly* tainted variables are those for which non-mergeable tainting paths may exist. For the strongly tainted variables a witness execution can be constructed.

We provide hereafter two heuristics that can be used to identify *strongly* tainted variables:

1. If a tainting path $\mathcal{P}$ contains uniquely events from the window's body and its tainting source variable is strongly tainted, then it defines a strongly tainted variable as well. Since $\mathcal{P}$ contains only instructions in body epoch it has no conflicts with paths in the succeeding window.

2. If a variable x is tainted in two consecutive epochs and (i) there is no kill of this variable in the common epoch and (ii) the variable that made it tainted in the first epoch is *strongly* tainted, then x is strongly tainted.

4.4 Respecting synchronization primitives

Synchronization mechanisms are widely used to impose an ordering between threads execution and ensure exclusive access to shared resources. We focus on the usage of mutexes for the synchronization of critical sections and take advantage of their semantics to infer more accurate *taint dependency paths* in our analysis. That is, we add extra restrictions to the $isValid(\mathcal{P})$ function.

We remind a mutex is a binary variable with states *locked* and *unlocked*. Critical sections are portions of code that should be executed atomically. To synchronize access to critical sections all threads need to acquire the necessary mutexes prior to entering their critical section, and release them once its execution is completed. A thread acquires/locks a mutex m by calling a blocking function $lock(m)$ and releases/unlocks it by calling $unlock(m)$. A successful call to $lock(m)$ allows the thread to enter the critical section and prevents other threads from entering their critical section protected by the same mutex m until it is released ($unlock(m)$) by the thread that initially obtained it. A mutex acquisition always has a matching mutex release. As mentioned earlier in section 3.1 synchronization events are also logged. We define hereafter a *protection* which is a triplet (m, e_l, e_u) where m is the mutex used to synchronize threads, e_l is the logged event corresponding to the locking of the mutex ($lock(m)$) while e_u is the event corresponding to the *matching* release of the mutex ($unlock(m)$). The locking event always precedes the unlocking event, thus $e_l \blacktriangleleft e_u$. Finally, we will use a dot notation in the sequel to refer to the elements of a protection. Thus, given a protection $p = (q, e_k, e_m)$ then $p.m = q$, $p.e_l = e_k$ and $p.e_u = e_m$.

A critical section may be synchronized using several mutexes. Moreover, the set of mutexes protecting events of a critical section may not be the same for all events. We call *context* the set of protections surrounding an event and define the function $cont(e)$ which returns the context for an arbitrary event e. More precisely:

$$cont(e) = \{\, p \mid p.e_l \blacktriangleleft e \ \wedge\ e \blacktriangleleft p.e_u \}$$

We provide hereafter a small example to clarify the notion of *protection* and *context* we just introduced. Listing 3 presents an excerpt of a log file with events produced by thread A. On the right side, we identify the two protections present in Listing 3 namely $p_a,\ p_b$ matched to mutexes m_a and m_b respectively. We also provide the context for events e_3^A and e_5^A.

e_1^A : $lock(m_a)$
e_2^A : $lock(m_b)$
e_3^A : `x = y;`
e_4^A : $unlock(m_b)$
e_5^A : `w = z;`
e_6^A : $unlock(m_a)$

Listing 3: Critical section

$p_a = (m_a, e_1^A, e_6^A)$
$p_b = (m_b, e_2^A, e_4^A)$
$cont(e_3^A) = \{\, p_a,\ p_b \,\}$
$cont(e_5^A) = \{\, p_a \,\}$

Moreover, we introduce two operators for contexts: $\sqcap$ which computes the set of mutexes shared between two contexts, and $\boxplus$ which computes the set of mutexes *used in same critical sections* and shared between two contexts. Finally, we define the function $Mutex(c)$ which gives the set of mutexes for context c.

$$c_1 \sqcap c_2 \;=\; \{m' \mid \exists (p \in c_1\,,\, q \in c_2) \;\text{ such that }\; p.m = q.m = m'\}$$

$$c_1 \boxplus c_2 \;=\; \{m' \mid \exists (p \in c_1\,,\, q \in c_2) \;\text{ such that }\; p = q \wedge m' = p.m\}$$

We note that the equality for two protections is defined as a complete match of all elements of the *protection* triplet:

$$p = q \;\Rightarrow\; p.m = q.m \;\wedge\; p.e_l = q.e_l \;\wedge\; p.e_u = q.e_u$$

$$Mutex(c_1) \;=\; \bigcup_{p \in c_1} p.m$$

4.5 Inferring order from mutexes

By definition of mutual exclusion critical sections protected by the same mutexes never execute concurrently. Thus, in the produced log files timestamps of synchronization events should be in accordance with the order they were executed. While the timestamps allow us to infer the exact ordering, we try to predict different serializations of entire critical sections if possible.

Figure 26 illustrates how the execution of two critical sections can be interleaved. In this instance we can clearly identify that the critical sections were executed in the order A, B. This ordering implies that only variable y should end up tainted. With the current slicing our analysis assumes all events are interleavable. Although the analysis should not interleave events belonging to a critical section (i.e. e_2^A, e_3^A, e_2^B, e_3^B), it can interleave the lock/unlock events resulting into considering a different synchronization where the ordering of critical sections is B, A. As illustrated in the summary $ST_{\mathcal{W}}$ our analysis predicts both serializations of the critical sections and thus both x and y are considered tainted.

In the example of Figure 26 the execution of critical sections could be interleaved. This is not always the case. For critical sections to be interleavable by the analysis they should appear within two consecutive epochs, i.e. be bounded in a window. If this is not the case then a fixed ordering between events in the critical section is imposed and it should be respected by the inferred *taint dependency paths*. For that reason we introduce the binary operator $p_a \lessdot p_b$ which checks if events protected by protection p_a precede those protected by p_b. The operator is defined as follows:

$$p_a \lessdot p_b \;\Rightarrow\; (p_a.m = p_b.m) \wedge \Big(\begin{array}{ll} Epoch(p_a.e_l) < Epoch(p_b.e_l) - 1 & \vee \\ Epoch(p_a.e_l) < Epoch(p_b.e_u) - 1 & \Big) \end{array}$$

Figure 27 illustrates the conditions for defining precedence based on the protections. The darkened areas are events in critical sections protected by the same mutex (let it be m) and the events surrounding it are the lock and unlock events. Each critical section defines a protection: $p_a = (m, e_k^A, e_m^A)$, $p_b = (m, e_k^B, e_m^B)$, $p_c = (m, e_k^C, e_m^C)$. We note that the lock release event (e_m^C) for critical section in thread C does not appear in the figure, but it definitively exists in a subse-

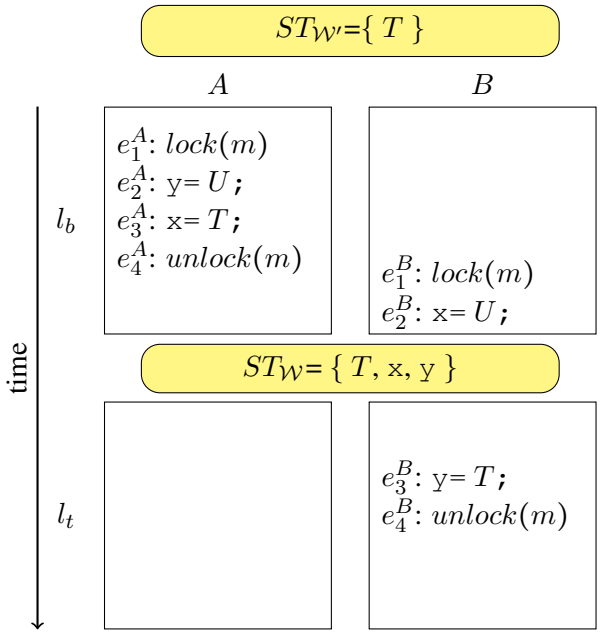


Figure 26: Interleaving critical sections

quent epoch. The relation $p_a \lessdot p_c$ obviously holds because the acquisitions of the mutex are not interleavable. Contrarily though, the acquisitions of the mutex are interleavable between critical sections of thread A and B. Despite that, the relation $p_a \lessdot p_b$ also holds because the acquisition event of thread A (e_k^A) cannot interleave with the release of mutex by thread B (e_m^B) which is necessary for swapping the execution order of the critical sections.

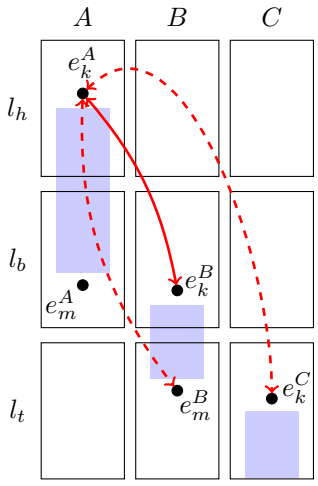


Figure 27: Ordering critical sections using mutexes

The precedence operator is also used to compare contexts $c_1 \lessdot c_2$. For a context to precede another we need to identify precedence between any two protections belonging to the contexts, that is:

$$c_1 \lessdot c_2 \;\Rightarrow\; \exists\,(\,p_a \in c_1\,,\, p_b \in c_2\,) \text{ such that } p_a \lessdot p_b$$

4.6 Enforcing explicit mutex ordering in taint dependency paths

When the critical sections cannot be interleaved, then their execution order must be respected by all inferred taint dependency paths. That for, we enforce the *consistency check* function ($isConsistent()$) by adding an extra restriction that enforces the precedence of events that constitute a taint dependency path based on their contexts:

Definition 4.7 (Predicate $isConsistent(\mathcal{P})$)

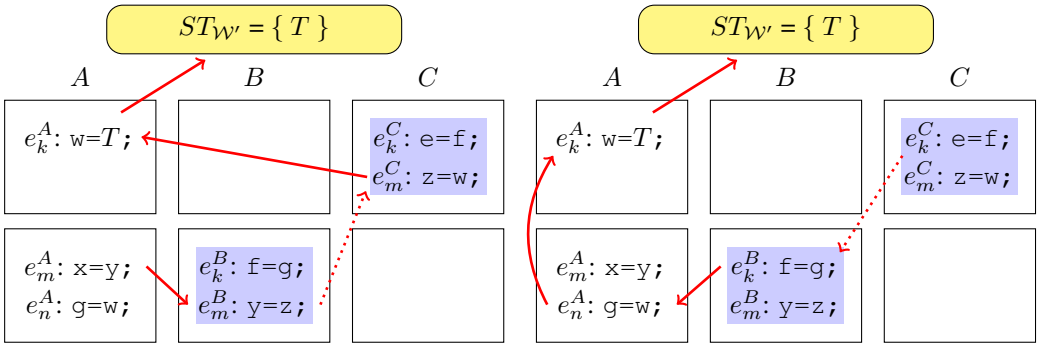
Ensures interleaving assumptions and explicit ordering imposed by critical sections are respected:

$$\forall e_k, e_m \in \mathcal{P} \text{ then } \begin{cases} k < m \Rightarrow e_m \lhd e_k \\ \qquad\qquad \bigwedge \\ cont(e_m) \neq \varnothing \wedge cont(e_k) \neq \varnothing \;\Rightarrow\; cont(e_m) \lessdot cont(e_k) \end{cases}$$

4.7 Enforcing implicit mutex ordering in taint dependency paths

As mentioned earlier critical sections can be re-ordered by the analysis when they reside within the same window. The order in which they are executed is defined by the taint paths containing events belonging to those critical sections. Once an order is set it should be respected throughout the path.

Figure 28 illustrates two instances of the same window each depicting a different taint dependency path imposing a different ordering of the critical sections. On Figure 28(a) the tainting path $\mathcal{P}_1 = (\, e^A_m, e^B_m, e^C_m, e^A_k \,)$ implies the execution order C, B of the critical sections since e^C_m precedes e^B_m. Dually, the tainting path $\mathcal{P}_2 = (\, e^C_k, e^B_k, e^A_n, e^A_k \,)$ in Figure 28(b) implies the execution order B, C of critical sections.



(a) Path defining C, B order (b) Path defining B, C order

Figure 28: Taint paths define implicitly order of critical sections

The examples provided above illustrate how a taint dependency path imposes the ordering of critical sections by visiting events inside them. Once the execution order is set between two (or more) critical sections we need to ensure that the path (i) is not bouncing between two critical

sections, protected by the same mutexes and (ii) the serialization of events respects the order imposed by the critical sections.

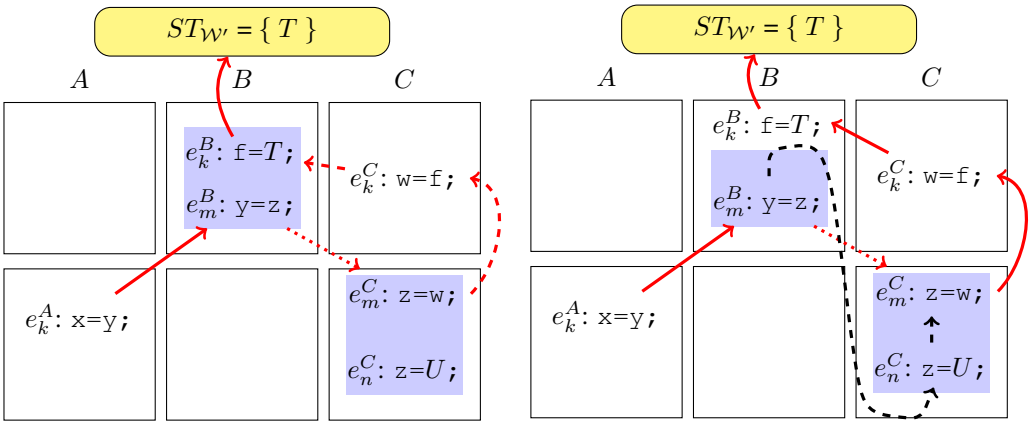
Figure 29(a) illustrates a path bouncing between two critical sections. That is, there exist two events in the tainting path that belong to the same critical section (e^B_m, e^B_k) and inbetween there exists an event that belongs to a mutually excluded critical section (e^C_m). In this example, initially events e^B_m, e^C_m connected with a dotted edge define the order C, B between the critical sections. Subsequently, events e^C_m, e^B_k linked with a dashed edge define the inverse ordering of critical sections (B, C). Thus the tainting path is no longer valid. To prevent such paths from propagating taintness we introduce a new path restriction $noRe - entry(\mathcal{P})$:

Definition 4.8 (Predicate $noRe - entry(\mathcal{P})$)

Ensures implicit ordering of critical sections is respected throughout the path:

$$\forall e_k, e_n \in \mathcal{P} \text{ such that } k < n \text{ then}$$

$$(\mathcal{M} = cont(e_k) \boxplus cont(e_n) \neq \varnothing) \Rightarrow \nexists e_m \text{ such that } Thr(e_m) \neq Thr(e_k) \wedge k < m < n \wedge Mutex(cont(e_m)) \cap \mathcal{M} \neq \varnothing$$



(a) Invalid path, redifing order of critical sections

(b) Serializing events respecting order of critical sections

Figure 29: Implicit precedence of critical sections

The second thing we need to take care of when considering the implicit ordering of critical sections, is to respect entirely the serialization of all events. Figure 29(b) has slightly modified the example of Figure 29(a) such that the bouncing between critical sections is avoided, but illustrates the problem of killing the linking variable (z in our example) on the serialization of the critical sections. Again the dotted edge connecting events e^B_m, e^C_m specifies the ordering of the critical sections to be C, B. Because the events belonging to different critical sections are explicitly connected (only one linking variable is used) all events on dashed path must be taken into account. More precisely we have to check there is no event that kills the linking variable.

To filter out these incorrect paths we add one more check called $atomicKill(\mathcal{P})$. It checks if there does not exist any kill (i.e. redefinition) of linking variable between two subsequent events of the path e_k, e_{k+1} that are executed by different threads and protected by a common mutex. The definition of $atomicKill(\mathcal{P})$ we provide takes into account the case of events protected by sets of mutexes (contexts). The events that should be checked are those preceding event e_k and succeed e_{k+1} and for which the set of protections is common for e_k and e_{k+1}.

Definition 4.9 (Predicate $atomicKill(\mathcal{P})$)

Ensures the linking variable between events belonging to separate threads and protected by a common set of mutexes is not killed by an event belonging in their serialization.

$$\forall e_k, e_{k+1} \in \mathcal{P} \text{ such that } (\mathcal{M} = cont(e_k) \sqcap cont(e_{k+1}) \neq \varnothing) \text{ then:}$$

$$\nexists e_m \text{ such that } e_m \blacktriangleleft e_k \ \wedge (cont(e_m) \boxplus cont(e_k)) \cap \mathcal{M} \neq \varnothing \ \wedge \ Def(e_m) = Def(e_{k+1})$$
$$\nexists e_m \text{ such that } e_{k+1} \blacktriangleleft e_m \ \wedge (cont(e_m) \boxplus cont(e_{k+1})) \cap \mathcal{M} \neq \varnothing \ \wedge \ Def(e_m) = Def(e_{k+1})$$

4.8 Recapitulation

In this section we addressed the problem of taint analysis for multithreaded programs, a representative information flow analysis widely used in vulnerability detection. We proposed an offline sliding window-based taint analysis which allows the prediction of taint propagations that could have occurred under valid serializations of the executed multithreaded program. We give hereafter an overview of our analysis and the refinements we did on taint prediction.

Online phase:

- *Unrestricted multithreaded program execution:* the program is executed without imposing any scheduling restrictions (i.e. it is not serialized) and memory accesses to both shared and thread local variables are logged.

Offline phase:

- *Slicing of log files into epochs:* such that events that were executed within a bounded time interval belong to the same or adjacent epochs.
- *Sliding window-based analysis:*
  - use a window of two consecutive epochs;
  - apply *taint prediction* on the window and create a summary that contains plausible taint propagations down to the analyzed window.

- *Taint prediction:* we use an *iterative algorithm* which allows predicting taintness propagation without enumerating all serializations of events in the analyzed window. The iterative algorithm is based on the equivalence between computing taint propagation and solving disjunctive boolean equation systems. The serializations inferred by the predictive algorithm can account for the memory model. We applied it to *sequential consistency* and made the following refinements:
  - safely untainting variables;
  - taking lock synchronizations into account.

Comparison to existing work

Existing works on dynamic information flow tracking of multithreaded programs force them to execute sequentially. This allows to apply typical dynamic information flow analyses as in the case of sequential programs. This somehow naive approach penalizes execution time of analyzed application but also eliminates the effects of weak memory models and simultaneous memory accesses. Moreover the analysis results are restricted to the serialized execution.

To the best of our knowledge the only works addressing the problem of dynamic information flow for parallel executions of multithreaded programs are those of Ganai et al. [GLG12] and Goodstein et al. [GVC+10]. We introduced these works in section 2.3.2. Having fully presented our work allows a closer comparison.

The work of [GLG12] is closer to ours since (i) it does not need specialized hardware and (ii) has an offline prediction phase for taint propagation between threads. The goal of their offline prediction phase is slightly different from ours since they try to predict the effect of different operating system schedules while we target mostly into predicting the effect for plausible serializations of the executed schedule.

A first point of comparison is on the runtime execution and information logged. In [GLG12] they perform thread local dynamic information flow at execution time and only log information on accesses to shared variables. When logging a write to a shared variable they also include the locally computed taintness such that it can be used for offline propagation. This choice leads to more concise logs but less precise predictions since not all propagations can be re-calculated offline.

The second point of comparison is the prediction algorithm. As mentioned above their logs are less precise thus taint propagations cannot be computed offline. Instead, they simply propagate taintness if there exists a write of a shared variable with a tainted value and a read of this variable by different threads. The order in which they were logged does not matter. Finally, they do not take untainting into account since they assume all interleaving of events to be plausible, at least in the implementation of $\text{DTAM}_{parallel}$. In the implementation of DTAM_{hybrid} they refine the propagation by taking into account happens before relations.

Regarding the work of Goodstein et al. [GVC+10] it is based on a specialized architecture and performs online prediction for usage in the context of lifeguards. The specialized architecture produces synchronization barriers among the cores. This architectural support is required to switch between program execution and program analysis. In addition it forms bounded blocks of code

executed in parallel. This corresponds to the notion of epochs we obtain by slicing offline the logs obtained by a parallel execution.

Their prediction mechanism is similar to ours [1] in that it uses a sliding window (consisting of three epochs instead of two in our case) and constructs summaries to capture the effect of inferred serializations. Their prediction methodology focuses on implementing classic dataflow analyses (e.g. available expressions) and thus consists into analyzing blocks independently and propagating necessary information between blocks. Concerning taint analysis, they state it is not straight forward to implement in their framework and give some elements on how to proceed. They provide, as we do, a taint prediction for a completely relaxed memory model and for sequential consistency. They also account for untainting to reduce false positives. Comparing the accuracy in taint prediction we have the benefit of taking synchronization mechanisms into account which they do not.

[1] our work is inspired from [GVC+10]

5 Implementation and experimentation

In this section we present the tool implemented and experimentations conducted to validate the theory presented in sections 3 and 4.

5.1 Proof of concept tool

As a proof-of-concept, we implemented a tool chain for our offline taint prediction analysis. The tool chain, presented in Figure 30, is split into an instrumentation phase (left side of figure) and execution and analysis (on the right side). First, the source files are instrumented to produce the log files. Next, the program is executed and log files are generated. The log files are sliced into arbitrarily sized epochs and taint analysis is performed as described in section 4.

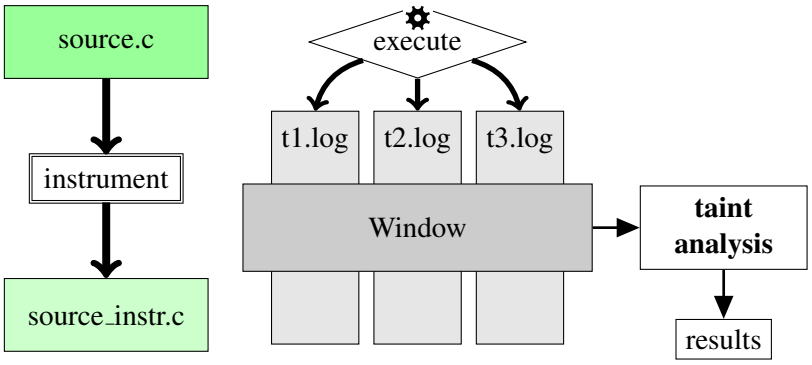


Figure 30: Abstract analysis framework

5.1.1 Source code instrumentation

The code instrumentation is implemented using the CETUS framework [DBM+09]. It is a C source-to-source compiler written in *Java* which provides some interfaces for analyzing and transforming the parsed C code. The instrumentation process consists in adding explicit logging instructions which record time-stamped information on used/defined variables of assignments. Special attention is given to keep track of variables passed as arguments into function calls and return values. Function calls related to mutex locking and un-locking are treated specially. For this proof of concept implementation not the entire ANSI C language can be instrumented. The limitations are purely syntactical and do not limit the analysis framework.

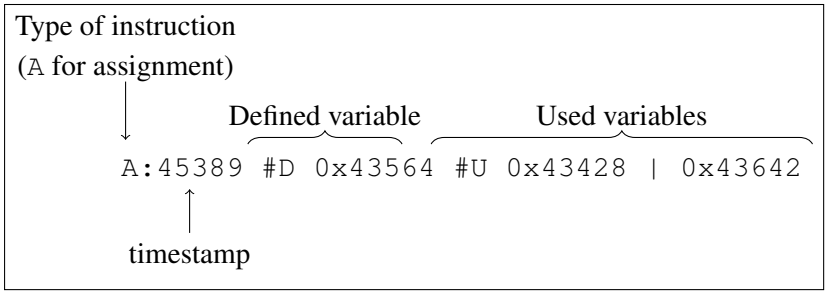
Each thread is logged in a dedicated file, so there is no need to synchronize logging instructions and thus we do not perturb much the applications execution. The time-stamping of log entries is in micro-seconds, relative to the beginning of the programs execution. The information carried by a log entry depends on the underlying instruction. In general it contains the set of *used* variables (actually their addresses on memory) and the *defined* variable (if it exists). Hereafter we sketch the instrumentation of some basic instruction types:

- *Assignments:* the *defined* and *used* variables are clearly stated. In the right hand side of assign-

ments there should be no function calls. For instance, the instrumentation of the assignment `x=y+z;` results into:

```
x=y+z;
fprintf(LFP,"A:%d #D %p #U %p | %p ", GET_TIME(),&x,&y,&z);
```

At execution time the log entry produced would look like that:



- *Functions:* passing arguments by value hides the dependency between the variables affected by the argument and the variable the function was called with. To overcome this problem we augment each function by adding a `void*` argument per variable in the argument list. Moreover inside the function we add a *dummy* assignment that will link the variable given as argument with the variable used inside the functions code during the offline analysis. Here is an example:

```
void f(int a){
  ...

void f(int a,void* aPT){
   fprintf(LFP,"A:%d #D %p #U %p ", GET_TIME(),&a,aPT);
   ...
```

The logging of function calls stores the time they were called, the function name and a set of variables that were passed as arguments. The only function that is time-stamped differently is `pthread_mutex_lock(..)` which timestamps the return of the function. This corresponds to the time the lock was obtained.

5.1.2 Log processing

Analyzing the log-files is divided into two phases. First, a slicer is used to set explicit *epoch boundaries* in the log-files. As mentioned in section 3.2 we use a time-based slicing. Second, the sliced log files are parsed and analyzed.

The parsing and main skeleton of the analysis are generic. They have been implemented in *Java* and are easily extendable. The parsing consists in reading from the log files time-stamped sets of used and defined variables. We read an epoch at time and feed the sliding window analysis with it. This implies that the log files do not need to be read entirely in memory to perform the analysis. The skeleton of the analysis (i.e. the *Vertical* and *Horizontal* passes) have been interfaced such that other analyses that can benefit from that structure can be easily implemented. Notably, one would implement a new analysis by re-defining: (i) what information is held in

the summary of an epoch (ii) how the analyzed property is propagated and (iii) how it is summarized.Finally, the framework also provides a lock-set analysis which identifies calls to library functions `pthread_mutex_lock` and `pthread_mutex_unlock` and encodes them as *protections* presented in section 4.4. This step should be performed *a priori* on the whole log files, since bounds of critical section may spawn over several windows.

At this time, the taint analyzer implementation makes no distinction between *strongly* and *weakly* tainted variables and can compute three types of taint propagation:

relaxed all interleaving of events in the window are accepted and kills are not taken into account;

sequential only sequentially consistent propagations are taken into account and variables killed are excluded from the summarization;

synchronized extends sequential propagation such that limitations introduced by locks are taken into account.

5.1.3 Visualizing taint propagations

Our tool is also capable of producing a representation of taint propagations in analyzed windows. We provide hereafter the visualization produced, during the analysis of a window with the three different types of analysis. The analysis has been slightly modified into that we produced all tainting paths that correspond to event $(18, 111, 5)$ in block $(18, 111)$. Each path is illustrated with a different color. Figure 31 illustrates the paths under *relaxed* analysis, Figure 32 under sequential and Figure 33 under synchronized.

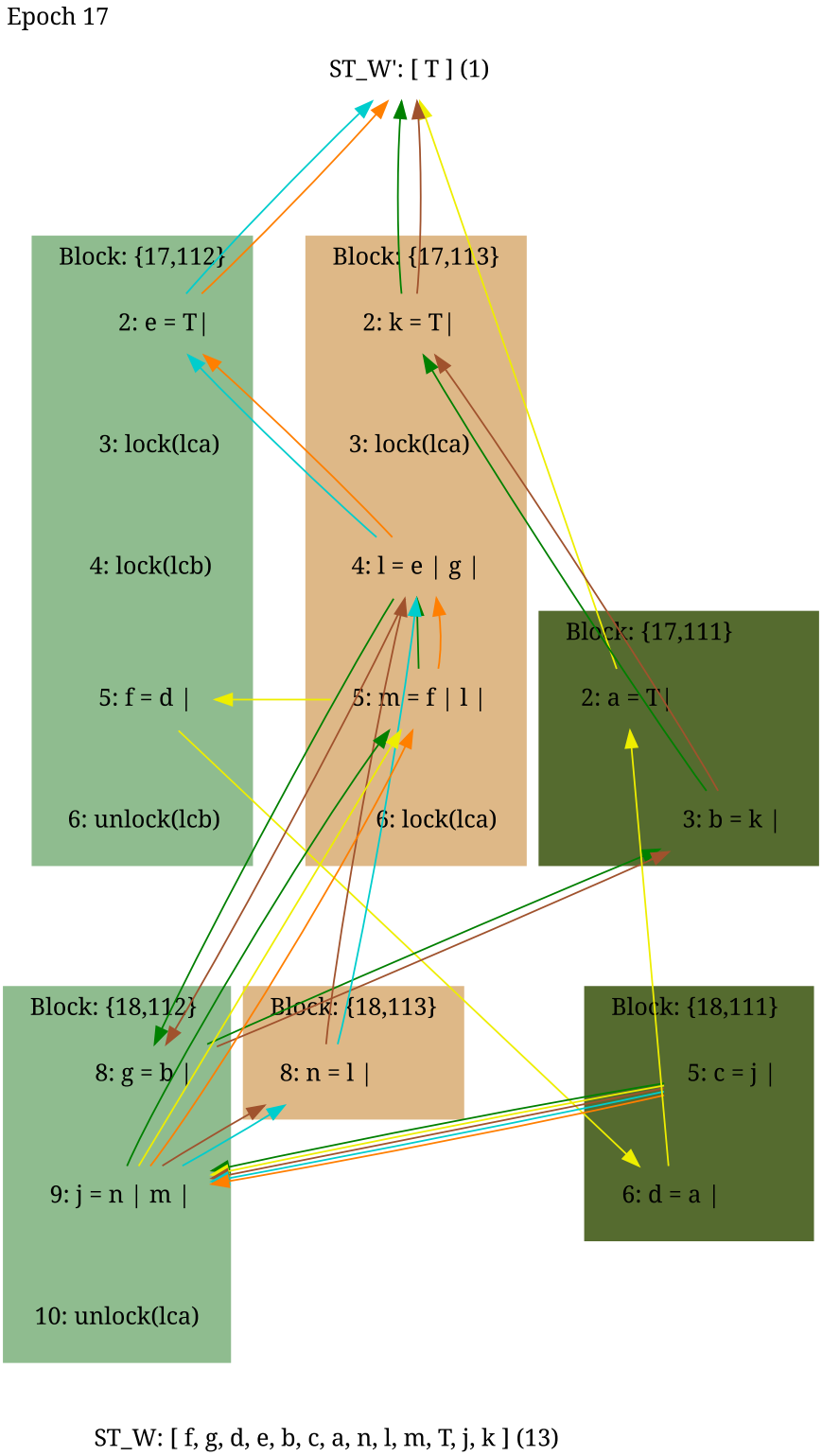


Figure 31: Tainting paths for $(18, 111, 5)$ under *relaxed* analysis

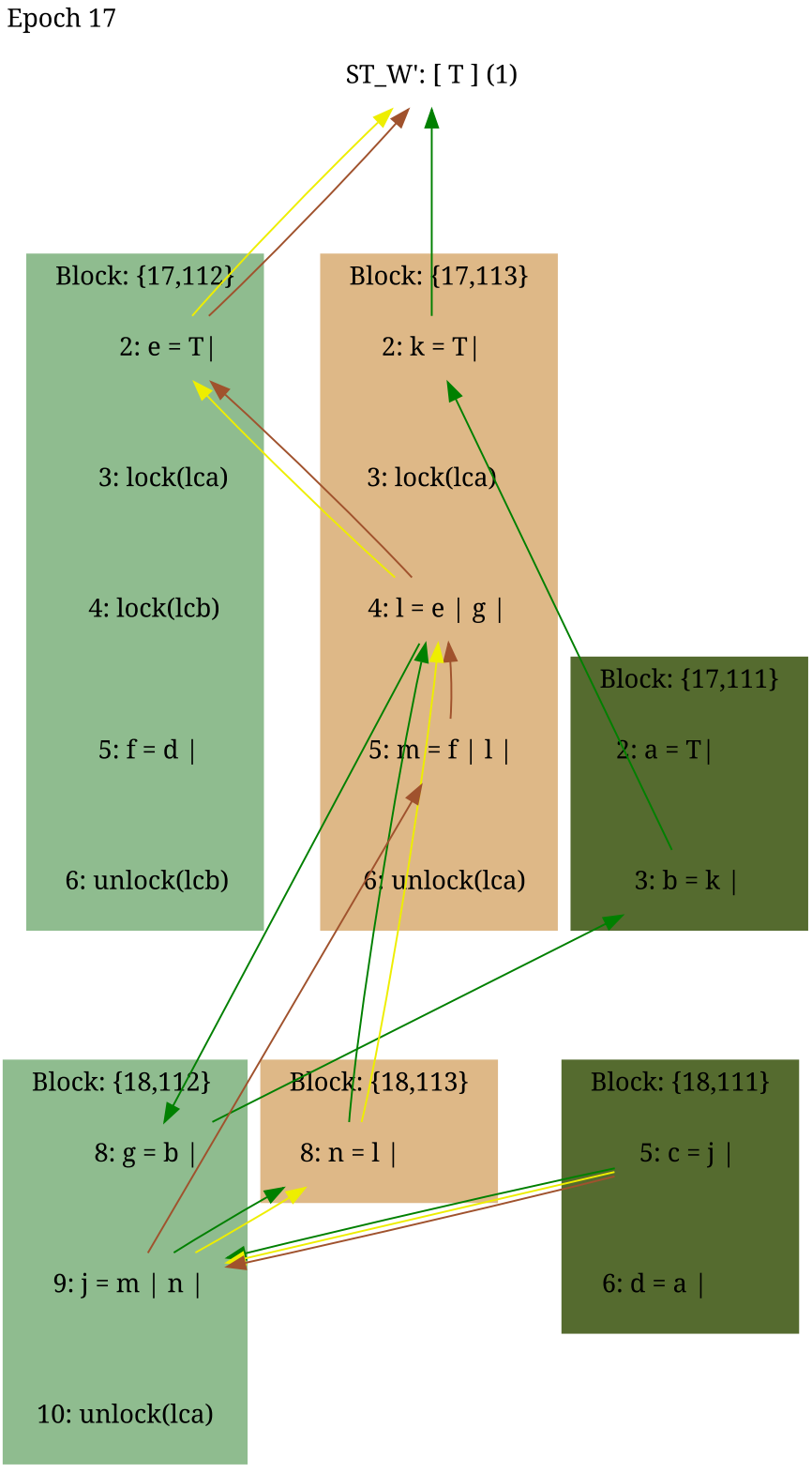


Figure 32: Tainting paths for $(18, 111, 5)$ under *sequential* analysis

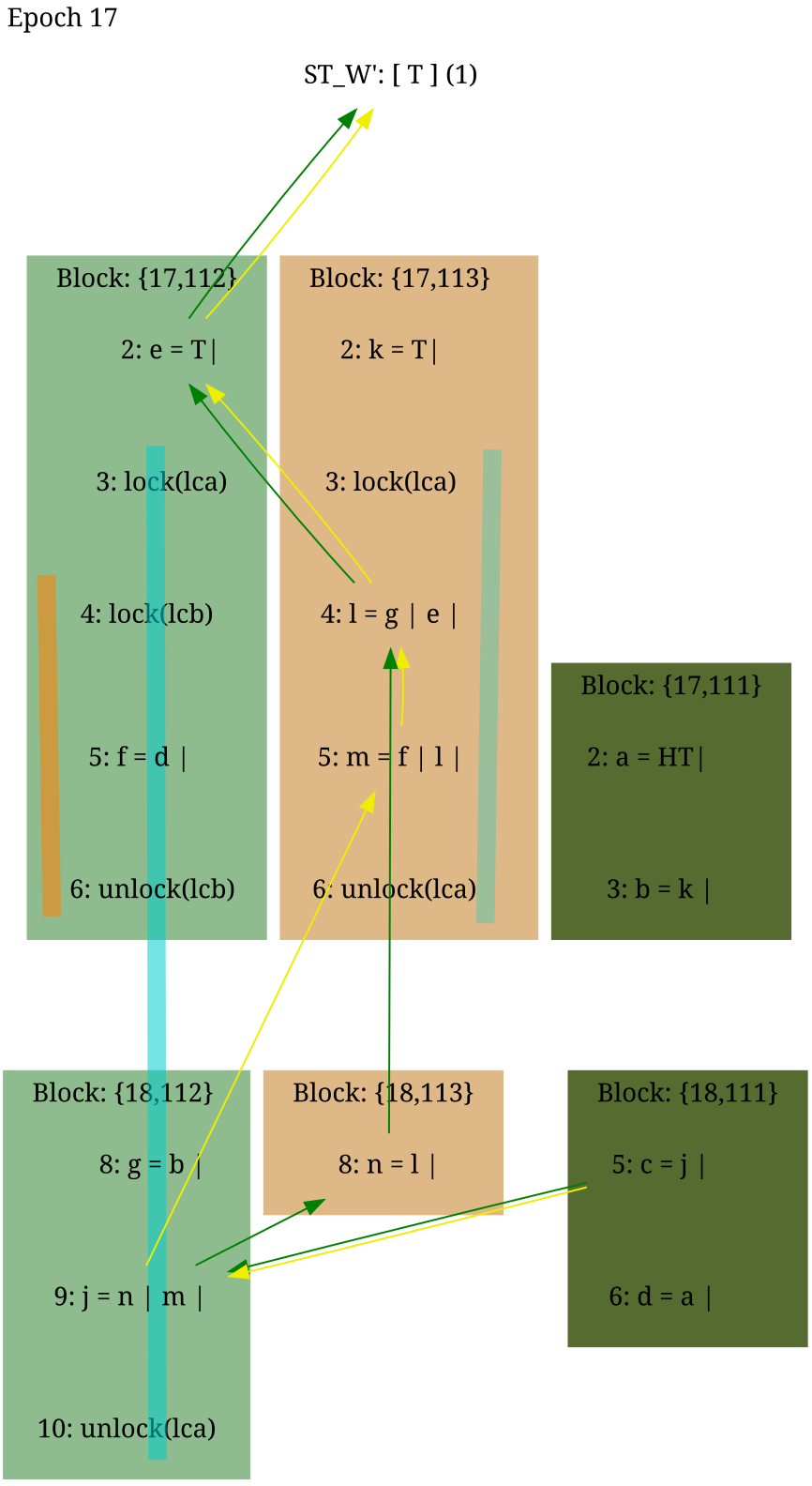


Figure 33: Tainting paths for $(18, 111, 5)$ under *synchronization* analysis

5.2 Some experimental results

To validate the framework and the analysis mechanics we initially used some toy examples (traceable manually), but rich enough to produce interesting behaviors especially regarding the epoch size. We illustrate those findings hereafter. Next,we applied the analysis on a bigger handcrafted example. In both cases, for simplicity the shared variables used are integers. The *printing* of a tainted variable is the malicious behavior we want to detect. The experimentations we carried out demonstrate: (i) the effect of epoch size on the accuracy of the analysis (ii) the reduction of false positives due to mutex support .

Thread A	Thread B	Thread C
1 `n=rand();`	1 `n=rand();`	1 `n=rand();`
2 `ack(n);`	2 `ack(n);`	2 `ack(n);`
3 `X=TAINT;`	3 `X=0;`	3 `print(X);`

The example above illustrates the code to be executed by distinctive threads. The function `ack` called is an *Ackermann* computation (time consuming) which adds non determinism in the order in which each thread executes its third instruction. Hereafter is the log produced by a parallel execution of the above program. During this execution the physical address corresponding to `X` was `&X=0x8b4` and the printed value was 0 (i.e. `X` was untainted).

thread A.log

```
A: 615 #D 0xb77     #U rand
F: 780 #F ack       #U 0xb77
A: 858 #D 0x8b4     #U 0x84f
```

thread B.log

```
A: 814  #D 0xb6f     #U rand
F: 878  #F ack       #U 0xb6f
A: 1108 #D 0x8b4     #U
```

thread C.log

```
A: 677  #D 0x24f     #U rand
F: 840  #F ack       #U 0x24f
F: 1752 #F print     #U 0x8b4
```

As we can observe in the log, Thread A taints shared variable `X` at time 858 (micro-seconds since the program started). Further in the execution at time 1108 Thread B un-taints `X` and consequently Thread C prints it at time 1752.

Figure 34 illustrates analyzing the log using two periodic partitionings. The solid and dashed horizontal lines denote the limit of epochs. Above each line/epoch we note the set of tainted variables observed by the analysis when entering that epoch. Using the left partitioning (bigger epoch size) an error is detected, since instruction `print(X)` of Thread C can precede the un-taint instruction of Thread B. On the contrary with partitioning used on right side (small epochs size) the printing is considered safe as based on the interleaving assumptions it cannot precede the un-tainting.

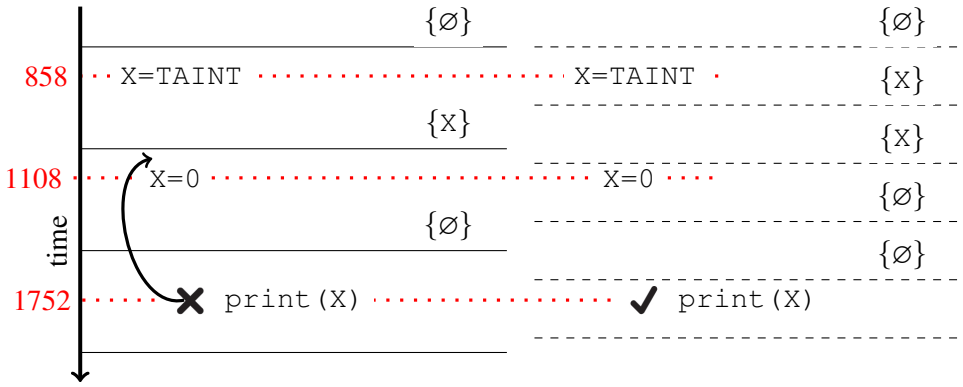


Figure 34: Cutting of epochs

The more complex hand-crafted example consists of a shared array of five elements which is randomly accessed by five threads. Each access is either: (i) an update with a random value (ii) an explicit taint (iii) an untaint operation followed by a print (iv) an update using another element of the array (to create longer taint dependency paths).Two variations of the application were tested. In the first one accesses are not synchronized and thus data races are likely to arise resulting into printing tainted variables. In the second, all accesses are protected using a mutex per element of the array. In this case, no errors occur since un-tainting and printing of an element are atomic. All executions are performed on a machine with 4 cores.

Simply executing the version without synchronization reveals some errors showing that tainted values can be printed. As expected, applying our analysis allows to find *more errors*. Table 2 presents how the size of the epoch chosen for the analysis affects the number of errors found. The second column of the table displays the number of errors observed at execution time per array element. The last columns display the number of errors detected by our analysis depending on the epoch size. As we can note, increasing this size increases the number of errors found by the analysis. Conversely, reducing too much the epoch size leads to *false negatives*, i.e., real errors are missed for epochs of size 1 μseconds (the number of errors detected by our analysis is smaller than the ones detected at runtime).

Node	Runtime errors	Epoch size in μ sec				
		100	50	20	10	1
0	1	8	6	2	2	0
1	2	11	7	4	2	1
2	0	9	3	0	0	0
3	1	5	3	1	1	1
4	0	5	1	0	0	0

Table 2: Errors found vs epoch size

	100	50	20	10
locks ignored	25	12	2	0
locks into account	4	0	0	0

Table 3: Using lock information in the analysis

In the example with mutex synchronization no errors occur at runtime. Table 3 displays the number of errors detected by our analysis depending on the epoch size. Without taking mutex into account plenty of errors are found. When mutex synchronization restrictions are applied by the analysis the number of reported errors reduces but still they correspond to *false positives*, that can be eliminated by considering the diagnostics produced by the analysis. On these examples: (i) executing the instrumented version of the application introduces an overhead of about 50% (ii) log file analysis takes less than 1 second on a Intel i3 CPU @2.4GHz with 3GB of RAM.

6 Conclusion and perspectives

In this work we focused on taint analysis, a representative information flow analysis, and proposed an efficient algorithm for *offline prediction* of taintness. During the online phase that precedes, the multithreaded program is executed without any scheduling restrictions (i.e. it is not serialized) and a log of all memory accesses is produced. Our prediction technique consists into breaking the logs into *epochs* which are then processed using a *sliding window*. Taintness is predicted locally for every window and summarized. The summary produced is used as input to the next window. The prediction algorithm uses an *iterative* method to infer taint propagations without enumerating and analyzing all plausible serializations of events in the window. We presented in details how to predict taint propagations under sequential consistency. Moreover, we included refinements to taint prediction such that untainted variables are correctly excluded from window summarizations. We further refined taint predictions by taking into account the semantics of locks. Finally, we implemented a proof of concept tool.

6.1 Perspectives for predictive information flow analysis

Identification of information flows is a central issue in all vulnerability detection tools. Existing tools either do not deal with multithreaded programs or they force them to execute sequentially. Our prediction algorithm could be incorporated in such a tool to allow extend the verdicts to a set of plausible serializations for a given parallel execution. In the context of testing it could be used in combination with a fuzzer to increase coverage and guide tests towards interesting executions. Finally, as observed by the experimentations, epoch slicing strongly affects predictions. Heuristics could be proposed for defining the minimum epoch size, or to indicate interesting events to use as slicing points.

The tool developed can be considerably improved. First, the current implementation of the source to source transformation could be extended such that more complex *C* programs can be processed. Also a more interactive interface for the analysis of windows would make back tracking of information flows more comfortable. Notably we could implement the distinction between *strong* and *weak* taintness. This would allow to exhibit a concrete trace, spanning over several windows, that propagates taintness to variables designated as *strong*.

Finally, a promising direction would be to use some dynamic binary instrumentation framework for generating the log files. This would unleash the restrictions on input programs. It would also allow producing much finer logs, since we would log events at the assembly level and not the source code level as we do now. At this level of logging it makes sense to consider other memory consistency models such as *TSO*.

A Boolean equation systems

Hereafter we provide some background information on *boolean equation systems* (BES). The definitions and notations we introduce are from [Kei05, GK04] and are standard in the literature.

Definition B.1 (Boolean expression [Kei05])

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be a set of boolean variables. The set of boolean expressions over $\mathcal{X}$ is denoted by $B(\mathcal{X})$ and is given by the grammar:

$$\alpha ::= \bot \mid \top \mid x_i \mid \alpha \wedge \alpha \mid \alpha \vee \alpha$$

where $\bot$ stands for false*, $\top$ stands for* true *and $x_i \in \mathcal{X}$*

Definition B.2 (Syntax of boolean equation system [Kei05])

A boolean equation system $\mathcal{E}$ is of the form $\sigma_i x_i = \alpha_i$ where $\sigma_i \in \{\mu, \nu\}$, $x_i \in \mathcal{X}$ and $\alpha_i \in B(\mathcal{X})$

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

Note that:

- *all left hand sides of the equations are different*
- *all variables in the right hand side are from $\mathcal{X}$*
- *the σ sign is μ if the equation is a least fixed point or ν if it is a greatest fixed point.*

Definition B.3 (Boolean equation system standard form [Kei05])

A boolean equation system $\mathcal{E}$

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

is in standard form if, for all $i \in [1, n]$, the right hand side expression α_i is of the form $y \circ z$ or y where $\circ \in \{\wedge, \vee\}$ and $y, z \in \mathcal{X} \cup \{0, 1\}$

Definition B.4 (Boolean equation system alternation depth [Kei05])

Given a boolean equation system $\mathcal{E}$

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

its alternation depth is the number of variables x_i with $1 \leq i \leq n$ such that $\sigma_i \neq \sigma_{i+1}$

A boolean equation system $\mathcal{E}$ is alternation free *if its alternation depth is zero. That is, all equations compute the same fixed point.*

Definition B.5 (Variable dependency graph [GK04])

Let $\mathcal{E}$ be a disjunctive/conjunctive *boolean equation system*

$$\mathcal{E} \equiv (\sigma_1 x_1 = \alpha_1)(\sigma_2 x_2 = \alpha_2)\ldots(\sigma_n x_n = \alpha_n)$$

the dependency graph of $\mathcal{E}$ is a directed graph $G_\mathcal{E} = (V, E, \ell)$ where:

- $V = \{i | 1 \leq i \leq n\} \cup \{\bot, \top\}$ *is the set of nodes*
- $E \subseteq V \times V$ *is the set of edges such that, for all equations $\sigma_i x_i = \alpha_i$*
  - $(i, j) \in E$ *iff a variable x_j occurs in α_i*
  - $(i, \bot) \in E$ *iff false occurs in α_i*
  - $(i, \top) \in E$ *iff true occurs in α_i*
  - $(\bot, \bot), (\top, \top) \in E$
- $\ell : V \to \{\mu, \nu\}$ *is the node labeling function defined by $\ell(i) = \sigma_i$ for $1 \leq i \leq n$, $\ell(\bot) = \mu$, and $\ell(\top) = \nu$.*

Lemma B.1 (Solution of BES implies path existence [GK04])

Let $G_\mathcal{E} = (V, E, \ell)$ be the variable dependency graph of a disjunctive (respectively conjunctive) boolean equation system $\mathcal{E}$. Let x_i be any variable in $\mathcal{E}$ and let the valuation v be the solution of $\mathcal{E}$. Then, the following are equivalent:

1. *$v(x_i) = \top$ (respectively $v(x_i) = \bot$)*
2. *$\exists j \in V with \ell(j) = \nu$ (respectively $\ell(j) = \mu$) such that:*
   (a) *j is reachable from i, and*
   (b) *$G_\mathcal{E}$ contains a cycle of which the lowest index of a node on this cycle is j*

References

[BH00] Bryan Buck and Jeffrey K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, November 2000. 2.2, 2.2.1

[BKK10] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1871–1878, New York, NY, USA, 2010. ACM. 2.2

[BRRS10] Gilles Barthe, Tamara Rezk, Alejandro Russo, and Andrei Sabelfeld. Security of multithreaded programs by compilation. *ACM Trans. Inf. Syst. Secur.*, 13(3):21:1–21:32, July 2010. 2

[Bru04] Derek L. Bruening. Efficient, transparent and comprehensive runtime code manipulation. Technical report, 2004. 2.2.2

[CFC12] Maria Castillo, Federico Farina, and Alberto Cordoba. 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, pages 175–179, Washington, DC, USA, 2012. IEEE Computer Society. 2.2

[CKS+08] Shimin Chen, Michael Kozuch, Theodoros Strigkos, Babak Falsafi, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Olatunji Ruwase, Michael Ryan, and Evangelos Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 377–388, Washington, DC, USA, 2008. IEEE Computer Society. 2.2.3, 2.3.2

[CLO07] James Clause, Wanchun Li, and Alessandro Orso. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM. 2.2.2

[CM09] Maximiliano Cristia and Pablo Mata. Runtime enforcement of noninterference by duplicating processes and their memories. In *WSEGI*, 2009. 2.2

[CZYH06] Winnie Cheng, Qin Zhao, Bei Yu, and Scott Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *Proceedings of the 11th IEEE Symposium on Computers and Communications*, ISCC '06, pages 749–754, Washington, DC, USA, 2006. IEEE Computer Society. 2.2.2

[DBM+09] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42:36–42, 2009. 5.1.1

[DKK07] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 482–493, New York, NY, USA, 2007. ACM. 2.2.3

[DM06] Bruno Dutertre and Leonardo De Moura. The yices smt solver. Technical report, 2006. 2.3.1

[ESKK08] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Comput. Surv.*, 44(2):6:1–6:42, March 2008. 2.2

[GK04] Jan Groote and Misa Keinänen. Solving disjunctive/conjunctive boolean equation systems with alternating fixed points. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *Lecture Notes in Computer Science*, pages 436–450. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-24730-2_33. A, B.5, B.1

[GLG12] Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: Dynamic taint analysis of multi-threaded programs for relevancy. In *Proceedings of the 2012 ACM SIGSOFT symposium on Foundations of Software Engineering*, FSE '12, New York, NY, USA, 2012. ACM. 2.2.2, 2.3.2, ii, 4.8, 4.8

[GVC+10] Michelle L. Goodstein, Evangelos Vlachos, Shimin Chen, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 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, pages 257–270, New York, NY, USA, 2010. ACM. 2.3.2, 2.3.3, 3.3, 4.8, 4.8, 1

[HLC09] Kim Hazelwood, Greg Lueck, and Robert Cohn. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proceedings of the 2009 international symposium on Memory management*, ISMM '09, pages 20–29, New York, NY, USA, 2009. ACM. 2.2.1

[JNPS09] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag. 2.3.1

[Kei05] Misa Keinänen. Solving boolean equation systems. Research Report A99, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, November 2005. A, B.1, B.2, B.3, B.4

[KW10] Vineet Kahlon and Chao Wang. Universal causality graphs: a precise happens-before model for detecting bugs in concurrent programs. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 434–449, Berlin, Heidelberg, 2010. Springer-Verlag. 2.3.1

[LCM+05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 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, pages 190–200, New York, NY, USA, 2005. ACM. 2.2.1, 2.2.2

[LELS05] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: a dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association. 2.2

[NG09] Vijay Nagarajan and Rajiv Gupta. Architectural support for shadow memory in multiprocessors. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '09, pages 1–10, New York, NY, USA, 2009. ACM. 2.2.2

[NKWG08] Vijay Nagarajan, Ho-Seop Kim, Youfeng Wu, and Rajiv Gupta. Dynamic information flow tracking on multicores, 2008. 2.2.3

[NS05] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*. The Internet Society, 2005. 2.2.2, 2.2.2

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. 2.2.1, 2.2.2

[OPAGS11] Meltem Ozsoy, Dmitry Ponomarev, Nael Abu-Ghazaleh, and Tameesh Suri. Sift: a low-overhead dynamic information flow tracking architecture for smt processors. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 37:1–37:11, New York, NY, USA, 2011. ACM. 2.2.3

[QWL+06] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 135–148, Washington, DC, USA, 2006. IEEE Computer Society. 2.2.2

[RGM+08] Olatunji Ruwase, Phillip B. Gibbons, Todd C. Mowry, Vijaya Ramachandran, Shimin Chen, Michael Kozuch, and Michael Ryan. Parallelizing dynamic information flow tracking. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 35–45, New York, NY, USA, 2008. ACM. 2.2.3

[RVS+06] Ram Rangan, Neil Vachharajani, Adam Stoler, Guilherme Ottoni, David I. August, and George Z. N. Cai. Support for high-frequency streaming in cmps. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 259–272, Washington, DC, USA, 2006. IEEE Computer Society. 2.2.3

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997. 2.2

[SFM10] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: weaving threads to expose atomicity violations. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, FSE '10, pages 37–46, New York, NY, USA, 2010. ACM. 2.3.1

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. 2.2

[SKSP06] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. Heapmon: a helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. Res. Dev.*, 50(2/3):261–275, March 2006. 2.2.3

[SM06] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006. 2

[SMWG11] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt, editors, *MEMOCODE*, pages 99–108. IEEE, 2011. 2.3.1

[SSP08] Prateek Saxena, R Sekar, and Varun Puranik. Efficient fine-grained binary instrumentationwith applications to taint-tracking. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 74–83, New York, NY, USA, 2008. ACM. 2.2.2

[SV98] Geoffrey Smith and Dennis Volpano. 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, pages 355–364, New York, NY, USA, 1998. ACM. 2

[VS97] Dennis M. Volpano and Geoffrey Smith. A type-based approach to program security. In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '97, pages 607–621, London, UK, UK, 1997. Springer-Verlag. 2

[WCGY09] Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 23–32, New York, NY, USA, 2009. ACM. 2.3.1

[WG12] Chao Wang and Malay Ganai. Predicting concurrency failures in the generalized execution traces of x86 executables. In *Proceedings of the Second international conference on Runtime verification*, RV'11, pages 4–18, Berlin, Heidelberg, 2012. Springer-Verlag. 2.3.1

[WRS] Waddington, Roy, and Schmidt. Dynamic analysis and profiling of multi-threaded systems. 2.2

[WS06a] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 137–146, New York, NY, USA, 2006. ACM. 2.3.1

[WS06b] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, February 2006. 2.3.1

[ZCYH05] Qin Zhao, Winnie W. Cheng, Bei Yu, and Scott Hiroshige. Dog: Efficient information flow tracing and program monitoring with dynamic binary rewriting abstract, 2005. 2.2.2, 2.2.2

[ZJS+11] David (Yu) Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. Tainteraser: protecting sensitive data leaks using application-level taint tracking. *SIGOPS Oper. Syst. Rev.*, 45(1):142–154, February 2011. 2, 2.2, 2.2.2, 2.2.2