High-Precision Sound Analysis to Find Safety and Cybersecurity Defects

Daniel Kästner, Laurent Mauborgne, Stephan Wilhelm, Christian Ferdinand
AbsInt Angewandte Informatik GmbH. Science Park 1, D-66123 Saarbrücken, Germany

Abstract

In recent years, security concerns have become more and more relevant for safety-critical systems. Many cybersecurity vulnerabilities are caused by runtime errors, hence sound static runtime error analysis contributes to meeting both safety and security goals. In addition, for cybersecurity goals, often sophisticated data and control flow analyses are needed, e.g., to track the effects of corrupted values, or determine dependence on potentially corrupted inputs. A sound analysis can guarantee that neither control flow paths nor read or write accesses are missed, even in case of data or function pointer accesses. To be feasible for industrial use, a static analyzer must be precise, i.e., produce few false alarms, and it must be user-configurable to allow analyzing specific data and control flow properties. It must also support efficient alarm investigation to minimize the manual effort needed to review the findings of the analyzer. In this article we give an overview of novel extensions of the sound static analyzer Astrée to minimize the false alarm rate, and to support advanced data and control flow analysis by taint analysis and analysis-enhanced program slicing. We describe an application of Astrée's taint analysis framework to detect Spectre v1/1.1/SplitSpectre vulnerabilities. Astrée's program slicer can also be applied for alarm slicing, which can significantly reduce the manual effort of reviewing the analyzer findings. Practical experience is reported on industrial avionic and automotive applications.

1 Introduction

A failure of a safety-critical system may cause high costs or even endanger human beings. With the growing size of software-implemented functionality, preventing software-induced system failures becomes an increasingly important task. It becomes paramount when fail-operational behavior is required, which is the case for systems providing highly automated driving capability.

One particularly dangerous class of errors are runtime errors due to undefined or unspecified behaviors of the programming language used. Examples are faulty pointer manipulations, numerical errors such as arithmetic overflows and division by zero, data races, and synchronization errors in concurrent software. Such errors can cause software crashes, invalidate separation mechanisms in mixed-criticality software, and are a frequent cause of errors in concurrent and multi-core applications. At the same time, these defects also constitute security vulnerabilities, and have been at the root of a multitude of cybersecurity attacks, in particular buffer overflows, dangling pointers, or race conditions [5].

In the past, security properties have mostly been relevant for non-embedded and/or non-safety-critical programs. Due to increasing connectivity requirements (cloud-based services, car-to-car communication, over-the-air updates, etc.), more and more security issues are rising in safety-critical software as well. Security exploits like the Jeep Cherokee hacks [17] which affect the safety of the system are becoming more and more frequent. Because of the increasingly pervasive monitoring of personal data including location data or health information, confidentiality breaches in embedded systems like mobile phones, automobiles, or airplanes have to be considered increasingly critical. Furthermore, data leakage might also have impacts on safety, e.g., if administrator or maintenance passwords are leaked.

Static code analysis has evolved to be a standard technique in the development process of safety-critical software. It can be applied to show compliance to coding guidelines, and to demonstrate the absence of critical programming errors, including runtime errors and data races. Abstract interpretation is a formal methodology for semantics-based static program analysis [2] which can be applied to demonstrate the absence of code defects due to undefined/unspecified behaviors, including runtime errors and data races. Abstract interpretation supports formal soundness proofs, i.e., it can be proven that – from the class of errors under analysis – no error is missed. Sound static analyzers provide full control and data coverage and allow conclusions to be drawn that are valid for all program runs with all inputs.

Safety-critical software is developed according to strict guidelines which effectively reduce the relevant subset of the programming language used and improve software verifiability. As an example dynamic memory allocation and recursion often are forbidden or used in a very limited way. By using formal techniques like abstract interpretation on such software, proving the absence of programming defects like runtime errors becomes possible with very low false alarm rates: then, the corresponding safety risks and security vulnerabilities have been effectively eliminated. In addition, for cybersecurity – and for data safety [15] –, advanced data and control flow analyses are needed, e.g., to determine data and control coupling, to track the effects of corrupted values, or to determine dependence on potentially corrupted inputs. Here, a sound analysis can guarantee that neither control flow paths nor read or write accesses are missed, even when functions are called indirectly via function pointers, and data is accessed via pointers.

While soundness means that false negatives can be ruled out, the analyzer may still report false alarms. In general, if the false alarm rate of a (sound or unsound) analyzer is too high, the effort to review and classify the findings may be economically unsustainable. In this article we focus on the sound static analyzer Astrée for which the zero alarm goal has been an important design criterion [1, 8, 9] right from the start. The key idea is to be parameterizable with respect to its abstract domains: new domains can be added, and there is a global hierarchy in the implementation, which determines what kind of information each domain may access, and what level of abstraction they can use. As an example, we will give an overview of its most recent abstract domain, designed to automatically detects finite state machines and their state variables, which allows to disambiguate the different states and transitions by partitioning. Experimental results on real-life automotive and aerospace code show that embedded control software using finite state machines can be analyzed with close to zero false alarms, and that the improved precision can reduce analysis time.

The contribution of this article is to give an overview of the key mechanisms to enable precise and flexible sound static analysis for safety and cybersecurity properties, with the example of the analyzer Astrée.

The article is structured as follows: after a brief introduction of abstract interpretation in Sec. 2, we outline general design considerations for achieving high analysis precision in Sec. 3, give an overview of Astrée's novel finite state machine domain (Sec. 3.1) and summarize practical experience made on industrial software. To enable configurable deep data and control flow analyses (Sec. 4), Astrée has been equipped with a generic abstract domain for taint analysis, which is described in Sec. 4.1. It allows Astrée to perform normal code analysis, with its usual process-interleaving, interprocedural and memory layout precision, while carrying and computing taint information at the byte level. Any number of taint hues can be tracked by Astrée, and their combinations will be soundly abstracted. Based on this generic taint analysis framework we implemented an automatic detection of Spectre V1/V1.1/SplitSpectre vulnerabilities (cf. Sec. 4.2). To complement taint analysis, which is a forward analysis, Astrée also has been extended by a program slicer which performs backward analysis to identify the code parts and inputs relevant for a selected slicing criterion (cf. Sec. 4.3). While supporting analyzing safety- or security-oriented information flow in a program, it also allows for automatic alarm slicing, which improves usability of the analyzer. This is the subject of Sec. 5. Sec. 6 concludes.

2 Abstract Interpretation

The semantics of a programming language is a formal description of the behavior of programs. The most precise semantics is the so-called concrete semantics, describing closely the actual execution of the program on all possible inputs. Yet in general, the concrete semantics is not computable. Even under the assumption that the program terminates, it is too detailed to allow for efficient computations. *Unsound* analyzers may choose to reduce complexity by not taking certain program effects or certain execution scenarios into account. A *sound* analyzer is not allowed to do this; all potential program executions must be accounted for. Since in the concrete semantics this is too complex, the solution is to introduce a formal abstract semantics that approximates the concrete semantics of the program in a well-defined way and still is efficiently computable. This abstract semantics can be chosen as the basis for a static analysis. Compared to an analysis of the concrete semantics, the analysis result may be less precise but the computation may be significantly faster.

Abstract interpretation is a formal method for sound semantics-based static program analysis [2]. It supports formal correctness proofs: it can be proved that an analysis will terminate and that it is sound, i.e., that it computes an over-approximation of the concrete semantics. Imprecisions can occur, but it can be shown that they will always occur on the safe side.

The difference between syntactical, unsound semantical and sound semantical analysis can be illustrated with the example of division by 0. In the expression $x/0$ the division by zero can be detected syntactically, but not in the expression a/b. When an *unsound* analyzer does not report a division by zero in a/b it might still happen in scenarios not taken into account by the analyzer. When a *sound* analyzer does not report a division by zero in a/b, this is a proof that b can never be 0.

3 Minimizing False Alarms

Astrée [14][9] uses abstractions to efficiently represent and manipulate over-approximations of program states. One simple example of abstraction used pervasively in Astrée is to consider only the bounds of a numeric variable, forgetting the exact set of possible values within these bounds. However, more complex abstractions can also be necessary, such as tracking linear relationships between numeric variables (which is useful for the precise analysis of loops).

As no single abstraction is enough to obtain sufficiently precise results, Astrée is actually built by combining a large set of efficient abstractions. Some of them, such as abstractions of digital filters – and now of finite state machines –, have been developed specifically to analyze control-command software as these constitute an important share of safety-critical embedded software. In addition to numeric properties, Astrée contains abstractions to reason about pointers, pointer arithmetic, structures, arrays (in a field-sensitive or field-insensitive way). Finally, to ensure precision, Astrée keeps a precise representation of the control flow, by performing a fully context-sensitive, flow-sensitive (and even partially path-sensitive) inter-procedural analysis. Combined, the available abstract domains enable a highly precise analysis with low false alarm rates.

To deal with concurrency defects, Astrée implements a sound low-level concurrent semantics [12] which provides a scalable sound abstraction covering all possible thread interleavings. The interleaving semantics enables Astrée, in addition to the classes of runtime errors found in sequential programs, to report data races, and lock/unlock problems, i.e., inconsistent synchronization. The set of shared variables does not need to be specified by the user: Astrée assumes that every global variable can be shared, and discovers which ones are effectively shared, and on which ones there is a data race. After a data race, the analysis continues by considering the values stemming from all interleavings. Since Astrée is aware of all locks held for every program point in each concurrent thread, Astrée can also report all potential deadlocks.

Practical experience on avionics and automotive industry applications are given in [8][13][9]. They show that industry-sized programs of millions of lines of code can be analyzed in acceptable time with high precision for runtime errors and data races.

3.1 Analyzing Finite State Machines

One long-standing problem for static analyzers is the analysis of finite state machine implementations: as control flow is encoded in data values, the values of state variables must be very precisely known in order to determine the feasible control flow paths. Using legacy abstractions, such analyses were either very imprecise or very slow and memory demanding, sometimes both. Hence the required precision used not to be feasible, typically resulting in high false alarm rates.

Consider the code fragment in Fig 1. It implements the state machine described in Fig 2 whose state is represented by variable `state`.

One major difficulty on this simple code, is to recognize that in state 3, the pointer `p` always points to the variable `state`. Using standard abstract domains, the analyzer cannot distinguish between individual iterations of the endless loop, it can only compute invariants which hold for any possible loop iteration. That means that the analyzer will (correctly) determine that at

```
1   int *p; int state = 0;
2   while (1) {
3     switch (state) {
4       case 0:
5         if (event) state = 1;
6         else state = 2;
7         break;
8       case 1:
9         state = 3;
10        p = &state;
11        break;
12      case 2:
13        if (event) state = 0;
14        else state = 1;
15        break;
16      case 3:
17        *p = 4;
18        break;
19      case 4:
20        return;
21    }
22  }
```

Figure 1: Example C implementation of Fig. 2

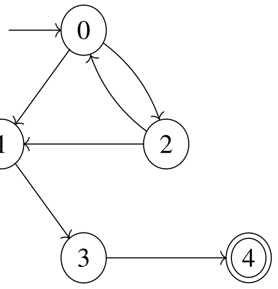


Figure 2: State machine

the beginning of any iteration of the loop *p* being either uninitialized (`invalid`) or pointing to `state`. After executing the statements of `case 0` the value of `p` did not change, but `state` can be in 1,2. After `case 1`, *p* points to `state`, and `state` must be 3. However, as the information at the beginning of the loop body is imprecise the analyzer cannot infer that when execution reaches `case 3`, it must have been in `case 1` in the previous iteration. Therefore it cannot exclude that `p` may be invalid and will raise an alarm in line 17 (possibly invalid pointer dereference).

3.1.1 The FSM Domain

The FSM (finite state machine) domain allows us to map each relevant program variable to an own abstract value for every possible value of a state variable in a finite state machine. This works by partitioning the abstract value of the memory domain according to the state variable.

Let us first assume we already have a basic abstraction for sets of memory states, which we assume to be sound and terminating. Such an abstraction introduces an abstract domain that will be referred in the following as the *underlying domain*. We denote this domain $\mathbb{D}^\#$, $\top$ its maximal element, $\sqsubseteq$ its approximation partial order, $\sqcup$ its join operator, and ∇ its widening operator (see Sec. 2). The meaning of an abstract element is described by the concretization function γ, such that for any abstract element a, $\gamma(a)$ is the set of memory states represented by a. This is summarized by the following notation:

$$\left(\mathbb{D}^{\#}, \top, \sqsubseteq, \sqcup, \nabla\right) \xrightarrow{\gamma} \left(\mathscr{P}(\mathbb{M}), \subseteq, \cup\right) \tag{1}$$

In order to formally define the state machine abstract domain, we introduce the sets: $\mathbb{L}$, of expressions of a program which are valid destinations of assignments (left-values), and $\mathbb{V}$, of integer values. Our goal is to define a new abstraction where we can have a separate abstract value of the underlying domain (i.e. a set of memory states) for each integer value that the state variable may take. For reasons of efficiency this partitioning must not be applied in cases where it is not relevant (i.e. when we are outside the scope of a finite machine implementation), or unnecessarily costly. Then the state machine domain $\mathbb{D}^{\#}_{\mathscr{A}}$ is defined as follows:

$$\mathbb{D}^{\#}_{\mathscr{A}} := \left(\left(\mathbb{V} \to \mathbb{D}^{\#}\right) \times \mathbb{L}\right) \cup \left(\mathbb{D}^{\#} \times \{\top\}\right) \tag{2}$$

An abstract value of the state machine domain is either a value of the underlying abstract domain combined with $\top$ to denote cases when we do not partition. Or it is a function $f^S : \mathbb{V} \to \mathbb{D}^{\#}$, called partitioning function, from integer values to abstract values in the underlying domain, combined with an expression $l \in \mathbb{L}$ that is currently partitioned. Here l is an lvalue of a state variable S, and the integer values in $\mathbb{V}$ are the values S can take. This allows us to have different abstract values in the underlying domain for each different value of S. As mentioned before, the underlying domain is the memory domain which keeps track of the values of all program variables. Hence we can separately keep track of the values of all variables for each possible value of the state variable.

3.1.2 Implementation

As mentioned above, Astrée combines a lot of abstract domains to achieve precision, parameterization and efficiency. There is a global hierarchy in the implementation, which determines what kind of information each domain may access, and what level of abstraction they can use.

The State Machine Domain builds on the Memory Layout Domain, which abstracts memory locations and pointer information, and associates memory locations to unique keys. All value domains only see the keys and compute sets of possible values for those keys. A dedicated helper value domain tracks which keys must be considered as state machine variables, and warns the State Machine Domain when such a variable is modified. Then, and only then, does the State Machine Domain trigger a call to **trans**, which is key to the efficient implementation for that domain.

While state machine state variables can be declared by an end-user using a simple directive (`__ASTREE_states_track((v));`), we always aim at the maximum automation of Astrée. In order to avoid this end-user declaration, we also have implemented automatic discovery of state machine state variables. It works in the following way:

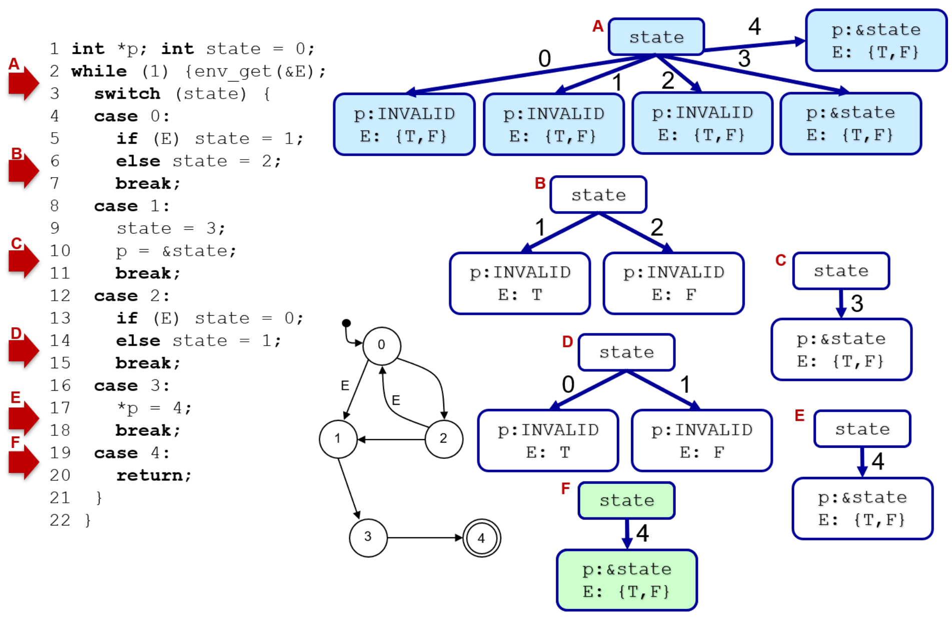
- Identify integer variables used in switch statements
- keep those variables which are also assigned in at least one case of the switch, possibly following function calls
- keep only the switch statements appearing inside possibly infinite loops

That heuristics captures some commonly generated state machines, and allows for a fully automatic and precise analysis of codes using this pattern. When the heuristics misses a state machine, the analysis will just miss the expected precision gain, which can still be achieved using directives. When the heuristics wrongly identifies a state machine, if the number of supposed states is small enough, the analysis will just be imprecise, and if the number of supposed states exceeds a built-in threshold, it will be rejected by the heuristics. Generalizing the heuristics to cover more complex patterns is subject of future work.

3.1.3 Applying the FSM Domain

The formal definition of the FSM Domain is given in [3]; here we will illustrate its operation with Fig. 3, which shows the analysis results of the working example of Fig. 1.

Let us call an abstract value of the underlying memory domain an *environment*. Using the heuristics described above, the analyzer automatically detects the finite state machine in the code and is aware that the variable `state` represents its states. Therefore it maintains a full partitioning of `state` which means that it keeps a separate environment for each possible value of `state`. The environment includes full information about the abstract value of all relevant program variables. In particular, this allows the analyzer to be aware of the fact that when `state` is 3 at the beginning of a loop iteration, `p` must point to `state`. Fig. 3 depicts this partitioning, i.e. the environments associated with every possible value of `state` at the beginning of the loop (A), immediately before the **`break`** statement in each switch case (B-E), and immediately before the **`return`** (F). More precisely, the tree structures of Fig. 3 represent the partitioning functions. For example, the tree shown at location (B) represents the abstract value $d = (f, \texttt{state})$, where `state` is the state machine variable, and f a partitioning function such that $f(1)$ is the abstract memory value (i.e. the environment) in which `state` is 1, `p` is *invalid* and `E` is *true*. $f(2)$ is the abstract memory value in which `state` is 2, `p` is *invalid* and `E` is *false*.



```
int *p; int state = 0;
while (1) {env_get(&E);
  switch (state) {
  case 0:
    if (E) state = 1;
    else state = 2;
    break;
  case 1:
    state = 3;
    p = &state;
    break;
  case 2:
    if (E) state = 0;
    else state = 1;
    break;
  case 3:
    *p = 4;
    break;
  case 4:
    return;
  }
}
```

Figure 3: Analysis result with novel FSM domain

3.1.4 Practical Experience

In [3] an detailed discussion of analysis results on several industrial examples is given. The largest benefit could be seen in an automotive control module implemented as hierarchal finite state machine composed of four sub-automata which comprises 348.530 lines of code. Without the FSM domain the analysis reports 45 alarms, needs 816 MB RAM and finishes in 24.5 minutes. With the FSM domain the analysis raises 4 alarms, finishes in 9 seconds, and consumes 424 MB RAM. The 41 alarms which are not reported any more have been confirmed as false alarms; the four remaining alarms can be considered justified. The reduction in analysis time and memory consumption is due to the fact that by applying the FSM domain the analyzer can eliminate many infeasible control flow paths that otherwise had to be taken into account.

In general, the required analysis time typically increases, and the observed increase depends on the number of combined states possible at a given program location during the run of the program. Also the memory needed for the analysis typically increases when we do an analysis with state partitioning. In the benchmarks under investigation in [3] the maximum observed increase in RAM usage is 40%.

4 Data and Control Flow Analysis

Safety standards such as DO-178C and ISO-26262 require to perform control and data flow analysis as a part of software unit or integration testing and in order to verify the software architectural design. Investigating control and data flow is also subject of the Data Safety guidance [15], and it is a prerequisite for analyzing confidentiality and integrity properties as a part of a security case. Technically, any semantics-based static analysis is able to provide information about data and control flow, since this is the basis of the actual program analysis. However, data and control flow analysis has many aspects, and for some of them, tailored analysis mechanisms are needed.

Global data and control flow analysis gives a summary of variable accesses and function invocations throughout program execution. It is the basis to determine the *data and control coupling*, as required by DO-178C.

In its standard data and control flow reports Astrée computes the number of read/write accesses for every global or static variable and lists the location of each access along with the function from which the access is made and the thread in which the function is executed. The control flow is described by listing all callers and callees for every C function along with the threads (and, if specified, the core) in which they can be run. Indirect variable accesses via pointers as well as function pointer call targets are fully taken into account. Filtering allows determining the control and data flow per software component. Astrée also provides a call graph visualization enhanced by data flow information, which can be interactively explored.

More sophisticated information can be provided by two dedicated analysis methods: taint analysis and program slicing. *Taint analysis* is a forward analysis and can answer questions about program parts affected by reading corrupted input values. *Program slicing* is a backward analysis which can answer questions about the program parts which might influence the value of a particular variable at a particular program point.

4.1 Taint Analysis

Astrée has been equipped with a generic abstract domain for taint analysis. It allows Astrée to perform normal code analysis, with its usual process-interleaving, interprocedural and memory layout precision, while carrying and computing taint information at the byte level. Any number of taint hues can be tracked by Astrée, and their combinations will be soundly abstracted.

Taint propagation can be formalized using a non-standard semantics of programs, where an imaginary taint is associated to some input values. Considering a standard semantics using a successor relation between program states, and considering that a program state is a map from memory locations (variables, program counter, etc.) to values in $\mathbb{V}$, the *tainted* semantics relates tainted states, which are maps from the same memory locations to $\mathbb{V} \times \{\text{taint}, \text{notaint}\}$, and such that if we project on $\mathbb{V}$ we get the same relation as with the standard semantics.

To define what happens to the *taint* part of the tainted value, one must define a *taint policy*. The taint policy specifies:

- **Taint sources** which are a subset of input values or variables such that in any state, the values associated with that input values or variables are always tainted.

- **Taint propagation** describes how the tainting gets propagated. Typical propagation is through assignment, but more complex propagation can take more control flow into account, and may not propagate the taint through all arithmetic or pointer operations.

- **Taint cleaning** is an alternative to taint propagation, describing all the operations that do not propagate the taint. In this case, all assignments not containing the taint cleaning will propagate the taint.

- **Taint sinks** is an optional set of memory locations. This has no semantical effect, except to specify conditions when an alarm should be emitted when verifying a program (an alarm must be emitted if a taint sink may become tainted for a given execution of the program).

Tainted input is specified through directives (`__ASTREE_taint((var;hues))`) attached to program locations. Such directives can precisely describe which variables, and which part of those variables, is to be tainted, with the given taint hues, each time this program location is reached.

Taint sink directives may be used to declare that some parts of some variables must be considered as taint sinks for a given set of taint hues. When a tainted value is assigned to a taint sink, then Astrée will emit a dedicated alarm, and remove the sinked hues, so that only the first occurrence has to be examined to fix potential issues with the security data flow.

The main intended use of taint analysis in Astrée is to expose potential vulnerabilities with respect to security policies or resilience mechanisms. Thanks to the intrinsic soundness of the approach, no tainting can be forgotten, and that without any bound on the number of iterations of loops, size of data or length of the call stack.

4.2 Detecting Spectre Vulnerabilities

Spectre/Meltdown attacks are transient instruction execution attacks, i.e., they exploit instructions which should not have an observable effect since they are speculatively executed [10] or have to be flushed because of an exception [11]. They affect a wide range of target architectures. The Spectre v1, Spectre v1.1 and Split-Spectre attacks (Spectre-PHT) are based on speculative execution, in particular, on branch prediction on array bound index checks. Taint analysis is particularly well suited to help detecting these vulnerabilities.

The first condition of Spectre-PHT vulnerabilities is the ability to control a variable through user (or public) input. The data flow from the corresponding input locations can be approximated using tainting, so first tainting directives have to be introduced, using the directive `__ASTREE_taint`. Typically this just requires tainting the arguments of some specific input functions.

The second condition is that such data controlled by the attacker are compared to a bound, so that speculative execution can be exploited. The idea here is to use the facility for Astrée to deal with more than one taint hue, to distinguish between possibly controlled, and possibly controlled and tested to be smaller than a bound. Since it would be quite demanding to manually add tainting directives for that to the source code under analysis, we added inside Astrée an automatic detection of comparison with bounds, which automatically changes the taint from *controlled* to *dangerous*.

Now the question is, how far in the code should variables stay dangerous? Speculative execution does not last forever, and in all known attacks so far, the memory access using dangerous variables must occur during speculative execution, which is one of the reasons why [16] introduced their speculative execution window. But we work on the source code level, and we aim at target architecture independence. One reasonable limit, though, is the length of the branches: when there is a test, there are two possible outcome, the branches, and when the control flow becomes the same whatever the outcome (the branches are merged) then the variable should not be considered dangerous anymore. The implementation challenge with that view is that tainting, by design, cannot be removed on joins. So, we came up with some non-standard use of the multi-hues tainting facilities offered by Astrée: we decided to taint public input with two hues (let us call them 1 and 2), and that flagging a memory location as dangerous consists in *removing* a hue (let us say it is hue 2). In that way, as long as the memory is tainted with only hue 1, it is considered dangerous, but as soon as we merge with a context where it is tainted with hue 1 and 2, it becomes merely controlled by the attacker again.

The third step is that the dangerous variable must be used to compute some memory address. Once again, we automatically discover in Astrée when a dangerous value is used to compute a memory location, and in that case, flag that address with a new taint hue. At each place where an address tainted with that hue is dereferenced, we emit a Spectre vulnerability alarm, and remove the tainting for that address, so that end-users can concentrate on the first occurrence, where they can, e.g., introduce fences that will anyway mitigate the vulnerability for all subsequent dereferences of the same address.

To illustrate the tainting algorithm we use the following example code shown in Figure 4:

```
volatile int controlled;
__ASTREE_volatile_input((controlled; [1,2]));
int victim_function( size_t x ) {
  if ( x < array1_size ) {
    temp &= array2 [array1[ x ] ] * 512];
  }
  return x;
}
void main(){
  unsigned int val, retval;
  init(&val);      //reads val from the environment
    __ASTREE_taint((val; controlled));
  retval = victim_function( val );
}
```

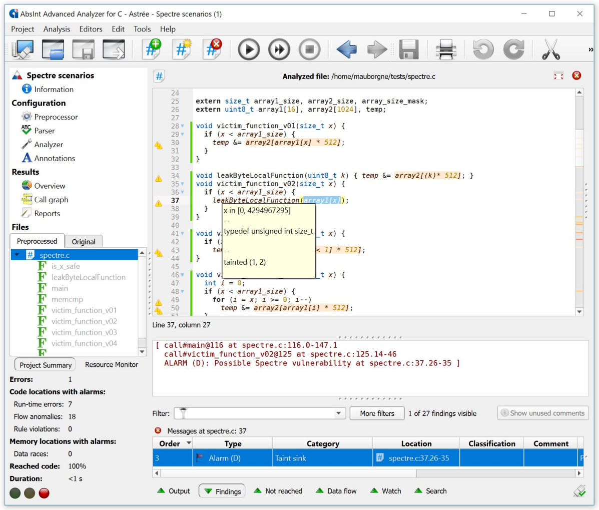
ALARM: Spectre vulnerability

Figure 4: Code excerpt with taint coloring.

In that code, `val` is tainted with hues 1 and 2, to denote that it may be controlled by the attacker. The taint is propagated to argument `x` of the victim function, and when `x` is compared to the size of the array, the tainting is transformed into hue 1 (hue 2 is removed from the tainting of `x`). This means that `x` is considered dangerous after the test. Then when `x` is used to compute an offset of `array1` before dereferencing, Astrée emits a Spectre vulnerability alarm.

A screen shot of a Spectre vulnerability alarm in Astrée is shown in Fig.5. The taint hues derived for variables in the code are shown in the tool tips.

This approach does not provide absolute safety from Spectre attacks. The first limitation is that tainting can only taint *reachable code*, and Spectre may be exploited on unreachable code (speculative execution may cause the execution of normally unreachable code). Note that Astrée displays unreachable code as parts of its normal output. If needed, the code can be made reachable to be covered by the analysis. Second, it targets a specific set of Spectre vulnerabilities, not all possible flavors of Spectre vulnerabilities. It embeds the Spectre detection into the runtime error analysis which is needed in safety-critical systems anyway, and reports vulnerabilities with



Analyzed file: /home/mauborgne/tests/spectre.c

```
extern size_t array1_size, array2_size, array_size_mask;
extern uint8_t array1[16], array2[1024], temp;

void victim_function_v01(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}

void leakByteLocalFunction(uint8_t k) { temp &= array2[(k)* 512]; }
void victim_function_v02(size_t x) {
  if (x < array1_size) {
    leakByteLocalFunction(array1[x]);
  }
}

void v                   _t x) {
  if (
    te                  < 1] * 512];
}

void v                   _t x) {
  int i = 0;
  if (x < array1_size) {
    for (i = x; i >= 0; i--)
      temp &= array2[array1[i] * 512];
```

```
x in [0, 4294967295]
--
typedef unsigned int size_t

--
tainted (1, 2)
```

Line 37, column 27

```
[ call#main@116 at spectre.c:116.0-147.1
  call#victim_function_v02@125 at spectre.c:125.14-46
  ALARM (D): Possible Spectre vulnerability at spectre.c:37.26-35 ]
```

Errors:	1
Code locations with alarms:	
Run-time errors:	7
Flow anomalies:	18
Rule violations:	0
Memory locations with alarms:	
Data races:	0
Reached code:	100%
Duration:	<1 s

Filter: 1 of 27 findings visible

Messages at spectre.c: 37

Order	Type	Category	Location	Classification	Comment
3	Alarm (D)	Taint sink	spectre.c:37.26-35		

Figure 5: Astrée GUI showing Spectre vulnerability alarm

high precision and on the basis of a sound analysis. This helps to significantly reduce the attack surface with little overhead.

The article [6] gives a detailed overview of Spectre attacks, Astrée's Spectre detection and its application to the PikeOS operating system. Astrée analyzes the whole code (400 000 LOC) in 2h30, using 17 GB of memory. During the analysis, Astrée does much more than warn about Spectre vulnerabilities, it also checks for compliance to coding rules, or warns about potential runtime errors. The precision is very high: Astrée only reports 68 locations with possible Spectre vulnerabilities, and manual inspection confirms that the false alarm rate is below 5%.

Further experiments were conducted on industrial avionics and automotive code. In the two cases described in the following we manually selected some global variables as taint sources since no information about actual user-controlled values was available to us. The first project is avionics software consisting of 2 million lines of preprocessed C code. It ran through in 2h43 (21 GB), compared to 2h36 without Spectre detection. The run with Spectre detection enabled found 113 possible vulnerabilities. The second project is an automotive application which consists of about 2.7 million lines of preprocessed C code. Without Spectre detection, it ran through in 1h42, and in 1h47 with Spectre detection enabled, and found 1271 vulnerabilities.

4.3 Program Slicing

The following definitions introduce the basic principles of static program slicing.

Definition 1 (Slicing Criterion) *Let P be a program. A slicing criterion in P is a tuple (s, V) which consists of a statement s and a set of variables V from P*

Definition 2 (Slice) *A slice S is a subprogram of P that exhibits the same behavior with respect to the slicing* *criterion* (s, V).

Computing *minimal* slices is an undecidable problem. However there are well-established algorithms for computing non-minimal, but still useful slices. A common approach is to compute a *System Dependence Graph* (SDG), which contains all data and control dependences of the program. Then a slice can be expressed as a reachability problem in this graph [4]. The precision of the slice directly depends on the precision of the SDG. However, computing precise system dependency graphs is a non-trivial task since it requires deriving intricate program properties. These may include points-to information for variable and function pointers, code reachability, context information or possible variable values at certain program points. As an example, over-approximating the set of possible destinations of a pointer variable blows up the size of the system dependence graph as it may add false dependences to statements which contain variables that would otherwise not be included in the slice. This may cause a drastic transitive increase in the number of dependences and vertices.

4.3.1 Analysis-Enhanced Slicing

A first novel contribution of Astrée's program slicer is that it leverages the invariants computed by the main Astrée analysis. In the following, we denote this slicing method *analysis-enhanced slicing*. This slicer can produce sound and compact slices by exploiting points-to and reachability information. Furthermore Astrée detects code which is guaranteed to be unreachable for any possible program execution. Ignoring such unreachable code fragments when constructing the system dependence graph further decreases its size. Hence, compared to slicing without leveraging Astrée invariants, a significant precision gain is achieved by reducing the amount of vertices and the amount of data- and control dependences in the system dependence graph. Another important advantage of analysis-enhanced slicing is its efficiency. While computing sound slices with standard static slicing requires lots of time and memory, those resources are significantly lower for analysis-enhanced slicing. This is due to the smaller size and smaller complexity of the computed system dependence graphs. This efficiency improvement makes it possible to compute slices for very large programs in feasible time.

A system dependence graph computed by our approach is a sound abstraction of the data- and control dependences of a computer program. This follows from the soundness of the Astrée core analysis. As a consequence, the resulting slices are also sound. The amount of precision gain depends on the precision of the exported invariants.

A detailed experimental survey of Astrée's analysis-enhanced program slicer with programs from automotive and avionic industry is given in [7]. It demonstrates that slicing can be applied to industry-size code with high precision and with feasible memory and computation time requirements.

5 Alarm Slicing

A second novel contribution of Astrée's slicer is its ability to compute a slice for one specific analysis context. The main application for this is *alarm slicing*: the goal is to compute a slice for the root cause of an alarm. By helping developers to prove the alarm to be a false alarm, or to recognize that it represents a true defect, respectively, the manual effort of reviewing the findings of the analyzer can be significantly reduced.

Astrée exhibits potential safety or security defects in the form of alarm messages. These messages are context sensitive with respective to the sequence of function calls and control decisions that lead to the particular alarm. Fig. 6 displays an example of such a context.

```
[ call#main at astree.cfg:18.0-50.1
  call#dhry_main at astree.cfg:40.6-17
  call#Proc0 at main.c:96.2-10
  loop=1/100 at Proc0.c:99.8-127.9
  loop=1/100 at Proc0.c:110.16-116.17
  call#Proc7 at Proc0.c:113.24-57
  ALARM (A): integer division by
        zero {0} at Proc7.c:78.12-22 ]
```

Figure 6: Context-sensitive alarm message of Astrée.

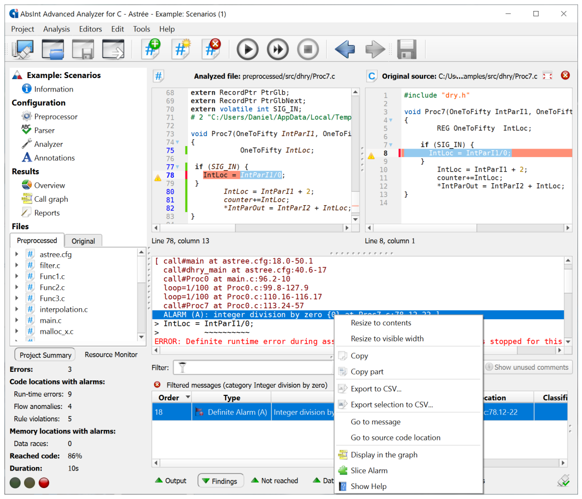
In order to compute precise slices for such alarm messages, the slicer has to take into account context information. The basic idea is simple: the critical situation for a division by zero alarm, which is reported for a given context, is precisely the context where the denominator becomes 0. Therefore we want to compute a partial slice for the variables used in the denominator, which only considers program paths leading to this particular context. Therefore, we augment the previously given definition of program slicing by call contexts.

Definition 3 (Call Context) *We denote a call context a sequence of functions* $[f_1, f_2, \dots, f_n]$ *in P, such that for each* $i \in \{1, \dots, n-1\}$ *it holds that* f_i *calls* f_{i+1}.

Definition 4 (Context-Sensitive Slicing Criterion) *A context-sensitive slicing criterion in P is a 3-tuple* $(s, V, [f_1, f_2, \dots, f_n])$ *consisting of a statement* $s \in f_n$, *s set of variables V and a call context* $[f_1, f_2, \dots, f_n]$ *of P.*

Definition 5 (Context-Sensitive Slice) *Let* $(s, V, [f_1, f_2, \dots, f_n])$ *be a context-sensitive slicing criterion and S a slice of P with respect to* (s, V). *A context-sensitive slice* S_c *is a subprogram of S, which behaves the same in the call context* $[f_1, f_2, \dots, f_n]$.

To compute context-sensitive slices we enhance the slicing algorithm given in [4] with a description of call contexts (stacks). In each step of the reachability analysis we additionally check that the dependences under examination match the relevant stacks. Dependences which do not match are discarded. When following a dependency edge which represents a function call, the



Analyzed file: preprocessed/src/dhry/Proc7.c

```
extern RecordPtr PtrGlb;
extern RecordPtr PtrGlbNext;
extern volatile int SIG_IN;
# 2 "C:/Users/Daniel/AppData/Local/Temp

void Proc7(OneToFifty IntParI1, OneToFi
{
            OneToFifty IntLoc;

 if (SIG_IN) {
   IntLoc = IntParI1/0;
 }
        IntLoc = IntParI1 + 2;
        counter+=IntLoc;
        *IntParOut = IntParI2 + IntLoc;
}
```

Line 78, column 13

Original source: C:/Us...amples/src/dhry/Proc7.c

```
#include "dry.h"

void Proc7(OneToFifty IntParI1, OneToFift
{
        REG OneToFifty  IntLoc;

    if (SIG_IN) {
        IntLoc = IntParI1/0;
    }
        IntLoc = IntParI1 + 2;
        counter+=IntLoc;
        *IntParOut = IntParI2 + IntLoc;
}
```

Line 8, column 1

```
[ call#main at astree.cfg:18.0-50.1
  call#dhry_main at astree.cfg:40.6-17
  call#Proc0 at main.c:96.2-10
  loop=1/100 at Proc0.c:99.8-127.9
  loop=1/100 at Proc0.c:110.16-116.17
  call#Proc7 at Proc0.c:113.24-57
  ALARM (A): integer division by zero {0}
> IntLoc = IntParI1/0;
>        ~~~~~~~~~~
ERROR: Definite runtime error during ass                s stopped for this
```

Errors:	3
Code locations with alarms:	
Run-time errors:	9
Flow anomalies:	4
Rule violations:	5
Memory locations with alarms:	
Data races:	0
Reached code:	86%
Duration:	10s

Filtered messages (category Integer division by zero)

Order	Type		cation	Classifi
18	Definite Alarm (A)	Integer division by	c:78.12-22	

Figure 7: Alarm Slicing in Astrée

topmost function is removed from the stack. This modification of the algorithm allows for context-sensitive slicing as defined above (considering exactly one call context), as well as context-insensitive slicing (considering all possible contexts). Thus it matches our definition of a context-sensitive slice which implies that for one program point a context-insensitive slice is identical with the union over all context-sensitive slices.

In our implementation, to ensure that only feasible call contexts are taken into account, and that calls-by-reference can be dealt with in a sound way, the slicer imports the intermediate internal program representation of Astrée. I then derives all possible call contexts from this representation. The soundness of the call contexts is directly inferred by the soundness of the Astrée analysis, in the same way as the information which is reused in analysis-enhanced slicing. The notion of context-sensitive slicing can be naturally extended to cover multiple call contexts.

In contrast to context-insensitive slices, context-sensitive slices do not capture all possible behaviors of the original program which influence the slicing criterion. Instead, the behavior described by the slice is restricted to execution paths which are in accordance with the set of considered call contexts. Context-sensitive slices tend to be significantly smaller than context-insensitive ones.

The different slicing modes presented in this section are relevant for demonstrating safety and security properties. Sound slices can be computed by context-insensitive analysis-enhanced slicing. With these slices it is possible to show that certain parts of the code or certain input variables might influence or cannot influence a program section of interest. They yield a global overview of these properties for the entire program.

In contrast to that, context-sensitive analysis-enhanced slicing, which only considers a subset of possible contexts, is more suitable for investigating the influence of a certain code section, e.g. a function, or a module, on the program location of the slicing criterion. Hence it is perfectly suited as a basis for automatic alarm slicing.

5.1 Automatic Slicing of Astrée Alarm Messages

The investigation of alarm messages which describe safety- or security issues of a computer program is an important task. In most cases defects only materialize in certain executions and thus dependent on call contexts and control decisions. Consequently, only these executions, and their respective contexts, are interesting for the alarm investigation. To simplify manual inspection each Astrée alarm is emitted in conjunction with problematic contexts. Unfortunately, even when utilizing the context information, alarm investigation is still a tedious and time consuming task. One main reason for this is the size of the code. Further, even with the additional information it is hard to identify which parts of the code are actually relevant for the issue under analysis.

Context-sensitive analysis-enhanced slicing is a natural approach to construct a more efficient workflow for alarm investigation. It can be configured to take into account the function call stack of the alarm context. On top of that, some of the other alarm context elements, in particular control decisions may be implicitly exploited with the imported analysis information, as described in Sec. 4.3. Hence, from the context menu of an alarm in the Astrée graphical user interface the computation of a slice for the alarm can be automatically triggered. In Fig. 7 this is displayed for the alarm and context of the example in Fig. 6. The Astrée program slicer then transforms the given context into a call context C by discarding all context entries that do not constitute function calls. Further, it determines the relevant set of program variables V that are part of the statement s for which the alarm is emitted. These steps yield the context-sensitive slicing criterion (s,V,C). By construction the resulting slice will behave the same as the original program for the examined context. Investigating that behavior is much more efficient, as the source code size is smaller and its content needs not to be filtered for relevance.

6 Conclusion

Sound static program analysis can be applied to demonstrate the absence of safety-relevant code defects, and of cybersecurity vulnerabilities. To make the application of such analyzers economically feasible, the false alarm rate must be low. Analyzing finite state machines at the C code level was a long-standing problem for static analyzers that typically lead to high false alarm rates. We illustrate a novel abstract domain which allows a highly precise analysis of finite state machines implemented in C programs. To support cybersecurity analyses, it is additionally necessary to support advanced data and control flow analyses, which allow to track the effect of value corruption in a forward and backward manner. We have developed a generic and user-parameterizable abstract domain for taint analysis and applied it to detect Spectre V1, V1.1 and SplitSpectre vulnerabilities. This is complemented by Astrée's novel program slicer, which supports precise backwards analysis by analysis-enhanced slicing. It can also be applied for automatic alarm slicing, making it easier for developers to determine whether a given alarm is justified. Experimental results on real-life software confirm both the precision and the efficiency of our approach.

Acknowledgment

This work was funded within the project PROFORMA by the German Federal Ministry for Education and Research with the funding ID 01IS17083A-C. The responsibility for the content remains with the authors.

References

[1] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety-Critical Software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4^{th} POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

[3] J. Giet, L. Mauborgne, D. Kästner, and C. Ferdinand. Towards zero alarms in sound static analysis of finite state machines. In A. Romanovsky, E. Troubitsyna, and F. Bitsch, editors, *Computer Safety, Reliability, and Security*, pages 3–18, Cham, 2019. Springer International Publishing.

[4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, Jan. 1990.

[5] D. Kästner, L. Mauborgne, and C. Ferdinand. Detecting Safety- and Security-Relevant Programming Defects by Sound Static Analysis. In J.-C. B. Rainer Falk, Steve Chan, editor, *The Second International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2017)*, volume 2 of *IARIA Conferences*, pages 26–31. IARIA XPS Press, 2017.

[6] D. Kästner, L. Mauborgne, and C. Ferdinand. Detecting Spectre Vulnerabilities by Sound Static Analysis. In R. F. Anne Coull, Steve Chan, editor, *The Fourth International Conference on Cyber-Technologies and Cyber-Systems (CYBER 2019)*, volume 4 of *IARIA Conferences*, pages 29–67. IARIA XPS Press, 2019. Archived in the free access ThinkMind^{TM} Digital Library, http://www.thinkmind.org/download.php?articleid=cyber_2019_3_10_80050.

[7] D. Kästner, L. Mauborgne, N. Grafe, and C. Ferdinand. Advanced Sound Static Analysis to Detect Safety- and Security-Relevant Programming Defects. In J.-C. B. Rainer Falk, Steve Chan, editor, *8th International Journal on Advances in Security*, volume 1 & 2, pages 149–159. IARIA, 2018.

[8] D. Kästner, A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, A. Schmidt, H. Hille, S. Wilhelm, and C. Ferdinand. Finding All Potential Runtime Errors and Data Races in Automotive Software. In *SAE World Congress 2017*. SAE International, 2017.

[9] D. Kästner, B. Schmidt, M. Schlund, L. Mauborgne, S. Wilhelm, and C. Ferdinand. Analyze This! Sound Static Analysis for Integration Verification of Large-Scale Automotive Software. In *Proceedings of the SAE World Congress 2019 (SAE Technical Paper)*. SAE International, 2019.

[10] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, Jan. 2018.

[11] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown. *ArXiv e-prints*, Jan. 2018.

[12] A. Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science (LMCS)*, 8(26):63, Mar. 2012.

[13] A. Miné and D. Delmas. Towards an Industrial Use of Sound Static Analysis for the Verification of Concurrent Embedded Avionics Software. In *Proc. of the 15th International Conference on Embedded Software (EMSOFT'15)*, pages 65–74. IEEE CS Press, Oct. 2015.

[14] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. *Embedded Real Time Software and Systems Congress ERTS^2*, 2016.

[15] SCSC Data Safety Initiative Working Group [DSIWG]. Data Safety (Version 2.0) [SCSC-127B]. Technical report, Safety-Critical Systems Club, Jan. 2017.

[16] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *CoRR*, abs/1807.05843, 2018.

[17] Wired.com. The jeep hackers are back to prove car hacking can get much worse, 2016.