Detecting Spectre Vulnerabilities by Sound Static Analysis

Daniel Kästner, Laurent Mauborgne, Christian Ferdinand
AbsInt GmbH
Email: info@absint.com
Science Park 1, 66123 Saarbrcken
Germany

Henrik Theiling
Sysgo AG
Email: office@sysgo.com
Am Pfaffenstein 14, 55270 Klein-Winternheim
Germany

***Abstract*—Spectre attacks are critical transient execution attacks affecting a wide range of microprocessors and potentially all software executed on them, including embedded and safety-critical software systems. In order to help eliminating Spectre vulnerabilities at a reasonable human and performance cost, we propose to build on an efficient industrial code analyzer, such as Astrée, which enables an automatic analysis of big complex C codes with high precision. Its main purpose is to discover run time errors, but to do so, it computes precise over-approximations of all the states reachable by a program. We enriched these states with tainting information based on a novel tainting strategy to detect Spectre v1, v1.1 and SplitSpectre vulnerabilities. The selectivity and performance of the analysis is evaluated on the embedded real-time operating system PikeOS, and on industrial safety-critical embedded software projects from the avionics and automotive domain.**

Keywords–Spectre; taint analysis; abstract interpretation; static analysis; embedded software; operating systems; safety; cybersecurity.

I. INTRODUCTION

In the past year, a series of cybersecurity attacks have been publicly reported, which exploit transient instruction execution, i.e., instructions which should not have an observable effect since they are speculatively executed or have to be flushed because of an exception [1] [2]. They have become known as Spectre or Meltdown attacks and they are highly problematic because they are rooted in hardware features present in most current hardware architectures. They can be considered confidentiality breaches: essentially, a malicious program can exploit Meltdown and Spectre to get hold of secrets stored in the memory of other running programs. Since the initial publication of [1] [2] there was a continuous stream of novel versions of transient execution attacks, so the full extend of the problem is still not fully known.

In the past, security properties have mostly been relevant for non-embedded and/or non-safety-critical programs. Recently 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 [3] 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 impact on safety, e.g., if administrator or maintenance passwords are leaked.

While Meltdown (so far) has only been reported on Intel and AMD processors, Spectre attacks affect a wide range of target architectures. As of today, four different classes of Spectre attacks have been reported, some of which comprise several distinct attack vectors. For some of the attacks, mitigation measures have been suggested that can be practically applied. The Spectre v1, Spectre v1.1 and SplitSpectre attacks are based on speculative execution, in particular, on branch prediction on array bound index checks. Vulnerabilities to these kinds of attacks can be discovered by static analysis. A naive mitigation strategy consists of flushing the cache or inserting memory barriers before every conditional instruction, which, however, would cause unacceptable runtime overhead. In our work we show that with low analysis effort it is possible to precisely identify Spectre v1/v1.1 and SplitSpectre vulnerabilities: there has to be an index bound check which depends on user-supplied data such that the accessed array element is used to access an element of another array. We will show that these vulnerabilities can be detected with very low false alarm rates, so that they can be safely mitigated with low runtime overhead.

The methodology we apply is abstract interpretation, a formal method for sound semantics-based static program analysis [4]. It supports formal soundness proofs (it can be proven that no error is missed) and scales to real-life industry applications. Abstract interpretation-based static analyzers provide full control and data coverage and allow conclusions to be drawn that are valid for all program runs with all inputs. Such conclusions may be that no timing or space constraints are violated, or that runtime errors or data races are absent: the absence of these errors can be guaranteed [5]. Nowadays, abstract interpretation-based static analyzers that can detect stack overflows [6] and violations of timing constraints [7] and that can prove the absence of runtime errors and data races [8] [9], are widely used for developing and verifying safety-critical software.

A. Related Work

Related work on applying static analysis to detect Spectre attacks has been reported in [10]. The difference to our approach is that [10] works on binary code, using mainly the BAP code analyzer [11], which cannot soundly analyze all possible behaviors, but relies on bounds to unroll loops, meaning that the approach cannot, in general, find all possible vulnerabilities. Our approach works at the source code level since here the analyzer can be sound and still be very precise about function pointer calls and other pointer accesses which reduces the false alarm rate, compared to approaches at the binary level. Another difference is that in our work we could also cover the SplitSpectre vulnerability which was made public only a few months ago. Also the code under analysis is different: we are focusing on real-life industrial code, which

 ISBN: 978-1-61208-743-6

is at the same time safety-critical and subject to cybersecurity requirements.

Taint analysis of big c programs was presented in [12], but it was based on a type system, and still a prototype that did not lead to an industrial quality tool.

B. Contributions

We discuss the impact of Spectre vulnerability on industrial running code, and explain how it can be mitigated. The target application used in our work is the real-time operating system PikeOS, which is used in aerospace, medical, automotive, railway, and industrial applications up to highest criticality levels. An operating system deployed in highly critical aerospace and automotive applications is subject to severe safety and cybersecurity requirements. We demonstrate that – without specific counter measures – even such a system is vulnerable to Spectre attacks and we show that by using our static analysis framework these vulnerabilities can be efficiently eliminated. Apart from the OS layer, we also evaluate our approach on real-life safety-critical applications from the avionics and automotive sectors.

As discovering the places to insert the mitigations is non-trivial (in fact, it can be proven to be undecidable), we propose to extend the sound static analyzer Astrée with a dedicated taint analysis, to help find a small number of places where mitigations should be considered. Our contribution includes the use of sets of taint hues, which allows to track complex taint states, and provides more precise hints for Spectre vulnerabilities. The precision and scalability of the tainting follows from the precision and scalability of Astrée on industrial code. Astrée runs on C programs, but the taint strategy presented in this article is applicable to other programming languages as well.

C. Overview

This article is structured as follows: in Section II we will give an overview of the Spectre vulnerabilities, focusing on the Spectre-PHT variants targeted by the approach presented in this article. Section III describes the kernel structure of the PikeOS operating system and discusses the basic concepts for mitigating Spectre vulnerabilities. After a brief introduction of static program analysis and abstract interpretation in Section IV, we describe the design and the work flow of the Astrée analyzer in Section V. Section VI starts with discussing static analysis of cybersecurity properties in general, then summarizes the key concepts of taint analysis, and concludes with a precise description of our taint analysis-based Spectre detection algorithm. The experimental results are presented in Section VII, Section VIII concludes.

II. SPECTRE

In this section, we give an overview of Spectre attacks with a focus on Spectre vulnerabilities and illustrate them with typical code examples. In doing so we follow the systematization from [13]. Spectre belongs to the class of transient execution attacks which use covert channels for transmitting data from transient execution stages to a persistent architectural state. Instructions whose execution has been started by the CPU but whose results are never committed to the architectural state are called transient instructions. They can occur due to out-of-order execution, speculation, but also due to exceptions and interrupts. Transient execution attacks transfer microarchitectural state changes caused by the execution of transient instructions to an observable architectural state. Spectre-type attacks exploit branch misprediction events; in contrast Meltdown-type attacks exploit transient out-of-order instructions following a CPU exception. To overcome the increasingly confusing naming of newly discovered transient execution vulnerabilities the article [13] proposes a naming scheme based on the microarchitectural element exploited by the attack:

- **Spectre-PHT**: V1 (CVE-2017-5753, Bounds Check Bypass) [14], V1.1 (CVE-2018-3693, Bounds Check Bypass on Stores) [15], and the newly discovered *SplitSpectre* [16] exploit the *Pattern History Table* (PHT)
- **Spectre-BTB**: V2 (CVE-2017-5715, Branch Target Injection) [14] exploits the *Branch Target Buffer* (BTB)
- **Spectre-STL**: V4 (CVE-2018-3639, Speculative Store Bypass) [17] exploits CPU memory disambiguation, specifically store-to-load forwarding (STLF)
- **Spectre-RSB**: ret2spec [18] and Spectre-RSB [19] exploit the Return Stack Buffer (RSB).

Note that Spectre V1.2 can actually be considered a Meltdown attack since it depends on a `#PF` exception [13].

In our work we are targeting Spectre-PHT since other vulnerabilies can be handled differently: Variant 2 can be fixed in one central point or by switching on compiler mitigations, so we need no analyzer. For V3 (Meltdown) and V3a, we expect a microcode update, and otherwise, software cannot protect against this anyway, so again, an analyzer does not help. For V4, if the microcode permits it, we can use the 'speculative store bypass disable'. Otherwise, V4 would need a binary analyzer, which is currently out of scope of this work. Similarly, for Spectre-RSB a binary-level analyzer would be required. The advantage of a source-level detection of Spectre V1/V1.1/SplitSpectre vulnerabilities is that it is possible to precisely and efficiently detect and mitigate them without the imprecisions and manual interactions to be considered for binary-level analysis.

The basic idea of Spectre v1/1.1 is to exploit that speculative execution of memory accesses modifies the cache in trusted code, which can then be detected from untrusted code using timing attacks, i.e., measuring timing of memory accesses to determine what the trusted code accessed. The vulnerability is a breach of security: the memory accesses that cause the cache to be modified in trusted code are not actually properly executed since they are protected by a range check. They are only speculatively executed, i.e., the effect of this speculative execution should have been discarded by the processor. But despite being executed only speculatively, the processor does not undo the effect on the cache. This, in effect, allows the timing attack to see the result of memory accesses that are properly protected by a check.

The scenario in which this vulnerability can be exploited is the following: there is trusted code that has two array accesses, the first of which is indexed based on data from untrusted code. This happens frequently in communication between trusted and untrusted code, e.g., with file handle ids, or other ids to address resources in trusted code. An example is given in Figure 1:

 ISBN: 978-1-61208-743-6

```
ErrCode vulnerable1(unsigned idx)
{
    if (idx >= arr1.size) {
        return E_INVALID_PARAMETER;
    }
    unsigned u1 = arr1.data[idx];
    ...
    unsigned u2 = arr2.data[u1];
    ...
}
```

Figure 1. Code with Spectre vulnerability.

In the code listing of Figure 1, `idx` is untrusted data that is used to index `arr1`. The array access is properly protected by an `if()` on the array size and generates an error message, so the code has no obviously broken security problems with the array access. However, the processor may still, despite the conditional, speculatively execute `arr1.data[idx]`. This speculative execution is discarded later, when the code exits the function with an error code, but the effect on the cache is not undone. Since the index is out of range, the array access can read much more memory that just the contents of `arr1`. Usually, the whole address space may be speculatively read by code like the above.

To read in user space what was the result of the access, the second array access comes into play: the value read from `arr1` is used to index `arr2`, so depending on the value of the first read, the second array access again modifies the cache. Untrusted code can then time which cache lines were touched to find out the value of `u1`, and since the index to `arr1` is under the control of untrusted code, untrusted code can effectively read any memory cell accessible in the trusted code.

All this is based on the speculative execution of the two array accesses, the result of which are discarded, but the cache effect is not undone, making this exploitable.

To fix this vulnerability in software, array indices in trusted code, no matter how well protected by a range check, must be fortified by mapping the array index into the array range. This limits the scope of the attack to the array itself, which is probably OK since with a value that is in range, trusted code would have accessed the array anyway. E.g., assume we have a function or macro `FENCEIDX` to map a value into $0..size-1$, we can rewrite the code from Figure 1 as shown in Figure 2 to protect against Spectre V1:

```
ErrCode vulnerable1(unsigned idx)
{
    if (idx >= arr1.size) {
        return E_INVALID_PARAMETER;
    }
    unsigned fidx =
        FENCEIDX(idx, arr1.size);
    unsigned u1 = arr1.data[fidx];
    ...
    unsigned u2 = arr2.data[u1];
    ...
}
```

Figure 2. Code from Figure 1 with protection against Spectre.

If the array access in this code is speculatively executed, the value of `u1` will be read only from `arr1`.

`FENCEIDX` can be implemented very efficiently on many architectures, so the impact on performance is negligible. The real problem is finding which array accesses to fortify, because a single missed vulnerable array access allows untrusted code to break into the trusted address space.

III. Protecting PikeOS against Spectre

PikeOS is SYSGO's embedded operating system and hypervisor. It is available for various 32-bit and 64-bit architectures: Intel and AMD x86, ARM, PowerPC, and SPARC. Some of these are affected by the Spectre vulnerability.

The PikeOS kernel was chosen for this work as a use case for examining how Spectre vulnerabilities can be detected with a static analyzer. The kernel is well suited because it is exactly functioning at different levels of trust, i.e., it receives via system calls information from untrusted user code and works with this information in the trusted kernel code. This is exactly the scenario where Spectre and similar vulnerabilities can be exploited. By being an operating system kernel for highly critical embedded systems, this is an interesting industrial use case.

A. Manual Spectre Mitigation

To examine how static analysis can help mitigate Spectre vulnerabilities in software, the PikeOS was first manually searched for patterns that are potentially vulnerable to Spectre V1. It turns out that the PikeOS kernel has only 700 array indirections, so manually checking each one for Spectre V1 is still feasible. When a potential vulnerability is identified, the counter-measure is to introduce a macro call that maps a user-provided array index to a value smaller than the array size, i.e., to rewrite `a[i]` as `a[FENCEIDX(i,n)]` where `n` is the array size. In the PikeOS kernel, we found 22 potential Spectre V1 vulnerabilities that we fixed this way. Different architectures implement `FENCEIDX(i,n)` in different ways to be as efficient as possible, and unaffected architectures just map it back to `i`.

The heuristics that was used for manual identification of a vulnerability may have caused false positive identifications, i.e., not all of these are guaranteed to be exploitable. This was done so that no vulnerability was missed by using an overly complicated identification strategy, i.e., we tried to make it easy for humans to judge the absence of a vulnerability in order to avoid mistakes. Whenever it looked like there could be one, the `FENCEIDX` was introduced. Note that a single failure to identify a vulnerability would leave the whole PikeOS kernel vulnerable.

The most recently documented SplitSpectre attack is most probably not applicable in PikeOS or any OS kernels, because the two parts of the exploit are distributed among trusted and untrusted code, separated by a system call layer in our case, where no value is leaking even on speculative paths. It is more likely to be exploitable in just-in-time compilation scenarios.

B. Kernel Code Structure

One complication for a static analyzer in the case of an operating system kernel is that the kernel is not a sequential application. So the typical static analyzer for application code will not be able to be applied directly the PikeOS kernel code.

 ISBN: 978-1-61208-743-6

On the other hand, the PikeOS kernel is a typical target for Spectre vulnerabilities, so adapting the static analyzer to the structure of an operating system kernel is another necessary step for a Spectre analyzer.

The structure of a kernel is typically split into two parts: the boot phase, where the kernel initializes data and hardware. This part is sequential, and the static analyzer needs this phase for its own boot-strapping of the values and contexts that the kernel executes in. In this phase, there is no user input, so no Spectre vulnerabilities can be found here. After the boot phase, the kernel basically turns into a library of callbacks that are triggered either by user-requests in the form of system calls, or by system events like interrupts. The system calls are the interesting ones for the Spectre analysis, as it is here where user provided data enters the PikeOS kernel.

IV. Static Program Analysis

Some years ago, static analysis meant manual review of programs. Nowadays, automatic static analysis tools are gaining popularity in software development as they offer a tremendous increase in productivity by automatically checking the code under a wide range of criteria. Here, the term *static analysis* is used to describe a variety of program analysis techniques with the common property that the results are only based on the software structure.

Purely syntactical methods can be applied to check syntactical coding rules as contained in coding guidelines, such as MISRA C [20], SEI CERT C [21], or Common Weakness Enumeration (CWE) [22]. They aim at a programming style that improves clarity and reduces the risk of introducing bugs. Compliance checking by static analysis tools has become common practice. Some of the rules require a deeper understanding of the code as they focus on semantical properties which requires knowledge about variable values, pointer targets etc.

To address such rules, and – even more importantly – to identify semantical code defects, semantics-based static analyses can be applied. Semantics-based methods can be further grouped into *unsound* vs. *sound* approaches, the essential difference being that in *sound* methods there are *no false negatives*, i.e., no defect will be missed (from the class of defects under consideration). Abstract interpretation is a formal method for sound semantics-based static program analysis [4]. 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 at 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.

V. Astrée

One example of a sound static runtime error analyzer is the Astrée analyzer [9] [23]. Its main purpose is to report program defects caused by unspecified and undefined behaviors according to the C norm (ISO/IEC 9899:1999 (E)). The reported code defects include integer/floating-point division by zero, out-of-bounds array indexing, erroneous pointer manipulation and dereferencing (buffer overflows, null pointer dereferencing, dangling pointers, etc.), data races, lock/unlock problems, deadlocks, etc.

The design of the analyzer aims at reaching the zero false alarm objective. For keeping the initial number of false alarms low, a high analysis precision is mandatory. To achieve high precision Astrée provides a variety of predefined abstract domains, e.g.: The interval domain approximates variable values by intervals, the octagon domain [24] covers relations of the form $x \pm y \leq c$ for variables x and y and constants c. The memory domain empowers Astrée to exactly analyze pointer arithmetic and union manipulations. It also supports a type-safe analysis of absolute memory addresses. With the filter domain digital filters can be precisely approximated, the interpolation domain tracks table lookups and interpolation functions. An automaton domain is available for precisely and efficiently analyzing finite-state machines. Floating-point computations are precisely modeled while keeping track of possible rounding errors.

Any remaining alarm has to be manually checked by the developers – and this manual effort should be as low as possible. Astrée explicitly supports investigating alarms in order to understand the reasons for them to occur. Alarm contexts can be interactively explored, the computed value ranges of variables can be displayed for each different context, the call graph is visualized, and a program slicer is available to identify the program parts contributing to a selected defect. By fine-tuning the precision of the analyzer to the software under analysis the number of false alarms can be further reduced.

To deal with concurrency defects, Astrée implements a sound low-level concurrent semantics [25] 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 [9] [26] [27]. 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.

VI. Taint Analysis-based Spectre-Detection

Taint analysis was first introduced as a dynamic analysis technique (e.g., in PERL), to try to find out which part of a code could be affected by some inputs. The original technique consisted in flipping normally unused bits, that would be copied around by operations and assignments. The same idea can be extended to static analysis by enhancing the concrete semantics of programs with tainting, the formal equivalent of the unused flipped bit in the dynamic approach. In the context of abstract interpretation, it is easy to abstract this

extra information in an efficient and sound way, using dedicated abstract domains. Conceptually, taint analysis consists in discovering data dependencies using the notion of taint propagation. 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 $\mathcal{V}$, the *tainted* semantics relates tainted states, which are maps from the same memory locations to $\mathcal{V} \times \{\text{taint}, \text{notaint}\}$, and such that if we project on $\mathcal{V}$ 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).

A. Taint Analysis in Astrée

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.

Tainted input is specified through directives 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. Any assignment is interpreted as propagating the join of all taint hues from its right-hand side to the targets of its left-hand side. In addition, specific directives may be introduced to explicitly modify the taint hues of some variable parts. This is particularly useful to model cleansing function effects or to emulate changes of security levels in the code.

The result of the analysis with tainting can be explored in the Astrée GUI via tooltips for all expressions appearing in the code, or explicitly dumped using dedicated directives. Finally, the 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. It seems particularly well suited to help detecting Spectre-PHT vulnerabilities, as these only occur in places where user input may interfere.

B. Detecting Spectre Vulnerabilities by Taint Analysis

The first step in Spectre-PHT vulnerabilities is to be able to control a variable through user (or public) input. Finding such variables can be approximated using tainting, so we first introduce tainting directives for identified public input. In the case of PikeOS, this is easily done, as the project analyzed by Astrée consists in one big loop with random calls to the OS: for each such call, we taint the parameters. In the code excerpt

```
void main(void)
{
  while (1) {
    switch (rand) {
      case 1:
        unsigned page;
        __ASTREE_initialize((page));
        __ASTREE_taint((page; controlled));
        os_call(page);
        break;
      ...
      case 148:
       int p;
       unsigned size;
       __ASTREE_initialize((p,size));
        __ASTREE_taint((p, size; controlled));
       os_call148(&p, size);
    }
  }
}
```

Figure 3. Example code with taint sources marked.

of Figure 3, `page`, `p`, and `size` are helper variables which are considered initialized with some unknown value. The Astrée directive `__ASTREE_initialize` models this effect and prevents alarms about uninitialized variable accesses. The directive `__ASTREE_taint` takes a comma-separated list of variables to be tainted and the taint hue to be used as parameters. The effect is that the system calls are analyzed with unknown, possibly attacker-controlled values.

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 [10] 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's call them 1 and 2), and that flagging a memory location as dangerous consists in *removing* a hue (let's 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.

It should be noted that we warn as soon as we find a single dereference of a dangerous address, whereas Spectre 1 requires two dereferences, but in practice that did not cause many false positives, and it seems that this criterion is necessary anyway to be safe with respect to SplitSpectre.

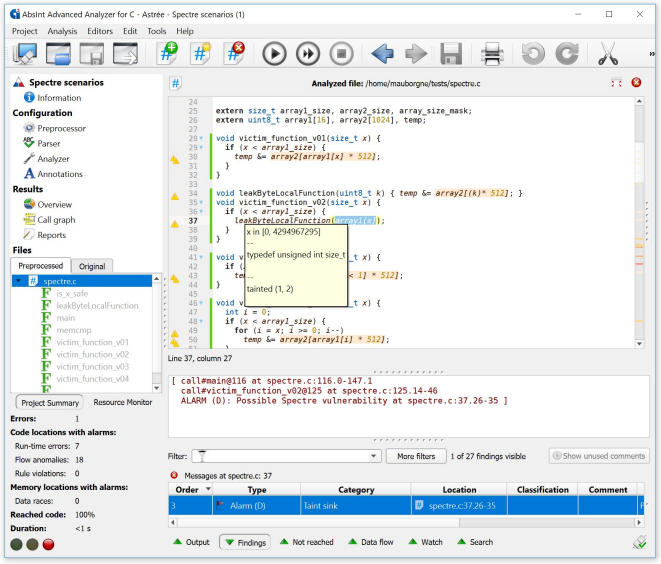

Figure 5. Astrée GUI showing Spectre vulnerability alarm

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

VII. Experiments

Our main experiment runs on the PikeOS sources, which are about 400 000 lines of preprocessed C code. Astrée can run with different levels of precision. Using a low precision level doesn't seem to hurt the precision of Spectre detection too much. In such mode, Astrée analyzes the whole code 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. But the extra precision needed for that analysis is not lost, as it allows the detection of Spectre vulnerabilities to be very targeted: Astrée only reports 68 locations with possible Spectre vulnerabilities.

A. Reviewing the Spectre Alarms

A very interesting aspect of that experiment is that PikeOS was already carefully analyzed by experts to root out all possible vulnerabilities. As expected due to Astrée's soundness,

 ISBN: 978-1-61208-743-6

all vulnerabilities found by the experts were also reported by Astrée. The precision of the analysis allowed the locations reported by Astrée to be reviewed in less than an hour. The review led to interesting conclusions:

- A number of false positives corresponded to places where the index used for the array access was shifted (using a right shift operation), so that even when the test for rang of the index failed, the final array access was always in range. In most case, the actual size of the array used in the code was not available to Astrée, so that Astrée could not have concluded that here were no vulnerability. It was a simple matter for the experts to assert that the right shift was enough to prevent a Spectre vulnerability, but we found that is was a good thing that such accesses be checked.
- A couple of alarms corresponded to calls from trusted code, so the data should be protected early enough, but it seemed to the expert that it would be a good idea to add fences for that cases anyway (which they hadn't before running Astrée).
- One interesting alarm exposed a possible Spectre 1.1 vulnerability. It was quite hard to discover, as the test on the input variable occurred two calls before its use in an array access. That makes it likely that the branch speculation would be finished before reaching the arrays access, but not impossible. It is one of those cases where a human can easily get lost with indirections and complex control flow, but where an automatic analyzer prevails.

B. Analysis Overhead

In order to asses the efficiency of our approach, we also ran an analysis of PikeOS without Spectre detection enabled, and finished only 5 minutes faster (2h25 instead of 2h30), using the same amount of memory.

In addition to the analyses of PikeOS we ran experiments on industrial avionics and automotive code. In both cases we manually selected some global variables as taint sources since no information about actual user-controlled values was available to us.

The avionics project consists 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 automotive project 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.

The immediate conclusion is that adding taint analysis in general, and Spectre detection in particular is quite costless for Astrée. Also, it seems that we found a good granularity for our detection criteria, since the number of findings is quite small with respect to the size of the code.

C. Further Experiments

For lack of time, we did not run the analysis of PikeOS with its mitigations yet, but that will be simple enough, as we will just include an Astrée untainting directive inside the fence macros used by Sysgo. This way Astrée will be able to confirm that the mitigation implemented in the code covers the Spectre attacks under consideration.

We also ran Astrée on simple code snippets on Spectre vulnerability published on Paul Kocher's web page [28], and Astrée shows the vulnerabilities in less than a second.

VIII. Conclusion

Spectre belongs to the recently discovered class of transient execution attacks which exploit common performance-enhancing microprocessor features. It can cause confidentiality breaches by leaking secret data through covert channels from transient execution stages to observable architectural states. It affects a wide range of microprocessors, including processors used for safety-critical embedded applications, trusted with particularly sensitive information.

In this article we have discussed the impact of Spectre on the safety-critical real-time embedded operating system PikeOS and outlined a mitigation strategy based on static taint analysis. We have presented a novel tainting strategy to detect Spectre V1, V1.1 and SplitSpectre vulnerabilities and discussed its implementation in the sound static analyzer Astrée. We have conducted experiments on the source code of the PikeOS operating system where the analyzer detects all vulnerabilities existing in the code while producing only few false alarms. Additional experiments on industrial avionic and automotive software confirm that the analysis is applicable to industry-size safety-critical application software at very little overhead.

Acknowledgment

This work was funded within the project ARAMiS II by the German Federal Ministry for Education and Research with the funding ID 01—S16025. The responsibility for the content remains with the authors.

References

[1] P. Kocher et al., "Spectre attacks: Exploiting speculative execution," ArXiv e-prints, Jan. 2018.

[2] M. Lipp et al., "Meltdown," ArXiv e-prints, Jan. 2018.

[3] Wired.com, "The jeep hackers are back to prove car hacking can get much worse," https://www.wired.com/2016/08/jeep-hackers-return-high-speed-steering-acceleration-hacks/ [retrieved: July 2019], 2016.

[4] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in Proc. of POPL'77. ACM Press, 1977, pp. 238–252. [Online]. Available: http://www.di.ens.fr/cousot/COUSOTpapers/POPL77.shtml[retrieved:July2019].

[5] D. Kästner, "Applying Abstract Interpretation to Demonstrate Functional Safety," in Formal Methods Applied to Industrial Complex Systems, J.-L. Boulanger, Ed. London, UK: ISTE/Wiley, 2014.

[6] D. Kästner and C. Ferdinand, "Proving the Absence of Stack Overflows," in SAFECOMP '14: Proceedings of the 33th International Conference on Computer Safety, Reliability and Security, ser. LNCS, vol. 8666. Springer, September 2014, pp. 202–213.

[7] J. Souyris et al., "Computing the worst case execution time of an avionics program by abstract interpretation," in Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis, 2005, pp. 21–24.

[8] D. Delmas and J. Souyris, "ASTRÉE: from Research to Industry," in Proc. 14th International Static Analysis Symposium (SAS2007), ser. LNCS, no. 4634, 2007, pp. 437–451.

[9] D. Kästner et al., "Finding All Potential Runtime Errors and Data Races in Automotive Software," in SAE World Congress 2017. SAE International, 2017.

[10] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via binary analysis," CoRR, vol. abs/1807.05843, 2018. [Online]. Available: http://arxiv.org/abs/1807.05843[retrieved:July2019].

[11] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, 2011, pp. 463–469. [Online]. Available: https://doi.org/10.1007/978-3-642-22110-1_37[retrieved:July2019].

[12] D. Ceara, L. Mounier, and M. Potet, "Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences," in Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings, 2010, pp. 371–380. [Online]. Available: https://doi.org/10.1109/ICSTW.2010.28[retrieved:July2019].

[13] C. Canella et al., "A systematic evaluation of transient execution attacks and defenses," CoRR, vol. abs/1811.05441, 2018. [Online]. Available: http://arxiv.org/abs/1811.05441[retrieved:July2019].

[14] P. Kocher et al., "Spectre attacks: Exploiting speculative execution," S&P, 2019.

[15] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," arXiv:1807.03757, 2018.

[16] A. Mambretti et al., "Let's not speculate: Discovering and analyzing speculative execution attacks," IBM Research Report RZ 3933 (#ZUR1810-003), Oct 2018.

[17] J. Horn, Speculative Execution, Variant 4: Speculative Store Bypass, https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 [retrieved: July 2019], 2018.

[18] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 2109–2122. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243761[retrieved:July2019].

[19] Microsoft Edge Team, "Mitigating speculative execution side-channel attacks in microsoft edge and internet explorer," https://blogs.windows.com/msedgev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer [retrieved: July 2019], Jan 2018.

[20] MISRA (Motor Industry Software Reliability Association) Working Group, MISRA-C:2012 Guidelines for the use of the C language in critical systems, MISRA Limited, Mar. 2013.

[21] Software Engineering Institute SEI – CERT Division, SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems. Carnegie Mellon University, 2016.

[22] The MITRE Corporation, "CWE – Common Weakness Enumeration," https://cwe.mitre.org [retrieved: July 2019].

[23] A. Miné et al., "Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée," in 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016), Toulouse, France, Jan. 2016.

[24] A. Miné, "The Octagon Abstract Domain," Higher-Order and Symbolic Computation, vol. 19, no. 1, 2006, pp. 31–100.

[25] A. Miné, "Static analysis of run-time errors in embedded real-time parallel C programs," Logical Methods in Computer Science (LMCS), vol. 8, no. 26, Mar. 2012, p. 63.

[26] 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). IEEE CS Press, Oct. 2015, pp. 65–74.

[27] D. Kästner et al., "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.

[28] Paul Kocher, "Spectre Mitigations in Microsoft's C/C++ Compiler," http://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html [retrieved: July 2019], 2018.

 ISBN: 978-1-61208-743-6