Fall 2020, Semester Project January 2020

Speculative control-flow hijack search based on compiler instrumentation

Andrés Sánchez

Abstract

Side-channel attacks can put in check existing software systems by leaking in atypical ways the information they process. It doesn't exist a way to ensure the total protection against side-channel attacks because of their nature of leaking information while respecting the semantics of the systems, moreover because there's not an assurance of all of them being pointed out. This project aims to bring closer the methodologies to detect and fix the classical software vulnerabilities and the side-channel-based ones, letting to ship software with a wich confidence of side-channel resilience. Focusing on speculative-based side-channel attacks, we propose a way to apply existing classical software vulnerability detection techniques to uncover possible Speculative Control-Flow Hijacks. We also demonstrate the suitability of this method in a real-world application, OpenSSH.

Professor: Mathias Payer

1 Introduction and motivation

Software is a computer system abstraction that defines the hardware expected behavior. In order to keep systems simplicity, the hardware defines a communication interface, its architecture. The microarchitecture is the processor's internal structure, which may vary for each processor responding to the same architecture. Still, the microarchitecture handles data in the processor intrinsics, which may happen to be revealed from the upper layers procedures.

These techniques revealing the information respecting the microarchitecture are denominated side-channel leaks. The architecture's semantics don't contemplate the microarchitecture's internal state, so its specific leak can be crafted in an architecturally correct way.

Any leak having its source from the microarchitecture can turn to be a more severe data leak, ending up in arbitrary code execution by the attacker by using ROP chains [1].

While classic software vulnerabilities mitigation methodology has received significant development. They help to disclose existing vulnerabilities and build more secure systems, side-channel research is in a different position. Side-channel vulnerabilities discovery is still in an early stage while they continue showing up, revealing new vector attacks over systems (which may happen to be difficult to exploit).

Among the side-channel vulnerabilities, the ones related to speculative execution are attractive because they're inherent to modern processors. Spectre type attacks[2] already demonstrated how subsequent memory accesses during a branch prediction which turns to be wrong can leave a trace in the cache, revealing the information with cache manipulation techniques.

There exist projects which aim to detect exploitable code under speculative execution, but none of them aim to have a realistic analysis of the threats that a program can present. This work brings closer the latent gap between classical software vulnerabilities and side-channel ones. We propose a way to use existing inspection methods for classical software vulnerabilities such as fuzzing to leverage the search of code snippets exploitable by speculative execution.

Current software vulnerabilities already are inspected under specific semantics that the architecture doesn't implement, e.g. memory safety. The techniques for their exploration can be applied as well to other "weird-semantics" problems such as side channels, as we do in this project.

Techniques to avoid side-channel exploitation can either be applied ahead in the hardware design (providing strong isolation of the information), with side-channel resistant software. If neither software or hardware are hardened against side-channel exploitation, an attack cannot be foreseen are avoided. By using the system performance counters and logging there can be an inference of unusual behavior of a potential attacker, but without demonstrating any kind of leak.

The case of Netspectre [3] is a reason why to look for these potential vulnerabilities, where an attacker can determine the memory contents without even sharing the system resources with the victim. In the case the victim shares resources with the attacker, the leak can be even broader, as an attacker has more potential side-channels and state observability too.

2 Background

2.1 Side-channels

Modern processors design focuses in the achievement of better performance metrics, which will take effect in most of the practical cases. As already mentioned, the optimizations will happen to be faster in a broaden set of cases, but because the complexity of the systems, it won't in all of them, so there will be some cases with a different outcome in the processor data flow. Based on the microarchitecture state and the observable differentiability from the architectural level, is possible to deduce the information which is influencing the computation. Side-channels fundament is the observable differenciability of the data being processed without illegal behavior.

For example, in the multiplication by exponentiation method, the multiplication argument bits can be determined because the duration of the operation. This case is enclosed on the branch decision ones: both outcomes reach the same state and the decision taken can be inferred; an observation on the time elapsed between the branch decision and the state where both outcomes meet, will determine which decision was taken.

2.1.1 Speculation-based side-channels

Modern processors have very different kind of instructions to use, each one with its own time to complete. Superscalar processors implement a pipeline to take advantage of all the functional units, whenever a conditional branch arises, the processor should hang the pipeline until the outcome of the condition is known or to make a prediction and rollback its effects in the case the prediction is wrong. More advanced architectures implement Out-of-Order execution to compute results in advance, even before other instructions have been resolved, and the results are committed only if all the previous instructions are committed, and their potential faults are not handled until they must commit because they may belong to a wrong speculated branch. In the case the instruction generating the condition of a branch has not been committed and the later branch prediction is wrong, all the subsequent instructions can produce exceptions in an internal state that they will not going to be reflected to the architectural state because the prediction rollback. As programs are not totally linear, they will depend on decisions that will make their control-flow to differ between executions.

This can solved by using predication, that removes all branches and keeps an uniform control-flow for all executions. Predication works by storing the outcome of a branch decission in a register, and based on that register value, it masks the values produced or have a branchless move of the results. This approach turns to be slow, as observable differenciability must be turned down, branchless approaches won't have the benefits of the hardware optimizations.

Branch prediction will produce the right branch to be predicted with a high statistically confidence. All the instructions inside a prediction are dependent on the outcome of the branch, and are discarded or committed depending on the condition outcome. The execution of the instructions during a wrong speculation are not reflected into the architectural state, but the effects they have performed in the microarchitectural state are not rolled back, e.g. the caches and other elements are modified during speculation unless there's a hardware mechanism that isolates the microarchitectural effects of speculated instructions.

If an observer can differ about the cache contents during speculation with a technique such as FLUSH+RELOAD, then it can reverse engineer the program's structure and leak information relevant to the speculated information.

2.2 Side-channel detection

Side-channels suppose a non-breaking semantics vulnerability of existing systems. Classical system vulnerabilities depend in a leaky implementation of the specifications to be exploitable. An exploit of the vulnerability must deliberately craft the flaws it presents (e.g. memory vulnerabilities).

To detect if a side-channel can be exploited, a further analysis from the correct semantics implementation is necessary.

However, the exploitation of a vulnerability is performed against the compiled program, which is produced based on the abstract definition in the source code level. To patch these type of vulnerabilities at the source code level can incur in a high complex definition, which can turn to be useful for the use of that system in a set of specific low-level systems (the hardware).

All the approaches have one objective in common: to leverage the underneath semantics of the operations that are not present in the architecture definition and can cause harm (and therefore are valid to be executed).

2.3 Static-analysis

This type of analysis is very useful to point out the vulnerabilities that can arise in a precise way, but it lacks of a mapping about how they can be exploited. It depends on a program to do the analysis over other one, by having an internal representation of the program analyzed and the instruction semantics. The outcome of the analysis is produced by checking program properties and showing the violation of the property in the case it exists. All the possible execution flows are contemplated and checked against the desired property the program has to meet.

2.4 Oblivious programming

To force the observable effects of a program to be indistinguishable during the sensitive information processing it exists the notion of *oblivious code*. Oblivious code generation can be applied either at the compiler pass which is generated or at the source of the program. To apply the instrumentation at the compilation level keeps the source-code simple while it can be applied at only certain architectures.

This hardening-focused approach imposes a constraint over all the different implementation models of a functionality to have the same observable effect.

Although it's effective to mitigate the issue, it can incur in a high performance drawbacks. For example constant time multiplication is the way to mitigate the timing-based side-channels in cryptography primitives which restricts the operation between any two variables to have the same delay. Another example, in the field of Spectre-like attacks, is to block all the memory operations until the speculation result has been resolved (by inserting LFENCEs) or either to substitute the operations that can perform a leak under speculation to don't do it for example with predication (speculative control-flow hardening).

2.5 Fuzzing under new semantics

An obvious approach to solve this problem is to include the hidden state of these operations in the architectural one, making it observable and traceable. To include this behavior needs from a modification of the processor model, so mainstream simulation techniques are not useful for it. This can be done either in a hardware device [4] for this purpose or by doing a simulation at software level[5]. However, the semantics at microarchitectural level are not contemplated in mainstream sanitizers and they may be included. Because the microarchitecture complexity, including the semantics of the operations would require a simulation of the whole processor at software level for this purpose, what's extremely costly.

3 Design

The methodology proposed to detect side-channel vulnerabilities depends on the source of an existing software is given.

Threat model: The threat model is based on a victim's program running in a shared domain with a potential attacker. The victim has an interface that the attacker can call, furthermore, the attacker can induce the branch predictor to a desired state, granting the ability to mispredict concrete branches even on the victim's execution.

The design follows an instrumentation to denote on which points of the software a side-channel observation can be triggered. According to them, generate the payloads that reach the specific points and depend on user input to produce a side-channel exploitation.

Althought we're going to look for a very concrete side-channel exploitation, the whole design generalization of side-channel leaks is divided in four orthogonal phases. Those phases can be implemented in a way that they are independent and with a minimal interface between them.

- **Semantics accommodation**: As already stated, the microarchitecture effects fall out of the architecture semantics and depend on the microarchitecture implementation. A simple example is how Out-of-Order processors can keep executing the next instructions from one with an illegal outcome, that produces an exception until that one doesn't reach the commit phase and its effects are not reflected in the execution. This phase defines the way that the semantics must be embedded in the existing code to inspect, it may fall under different ways to do it: Introduce the semantics in an interpreter that may be used later or by introducing the semantics in the binary without affecting the program's flow.
- **State exploration**: Given the way to interpret the program under the desired semantics, it can be explored whether they're broken or not either by producing different fixed traces (fuzzing) or with dynamic analysis.
- **Taint analysis**: Although the inputs have been generated, this phase needs to determine what the attacker capabilities are respecting breaking the semantics. This pass can be embedded in the state exploration as a constraint of the inputs generated.
- **State evaluation**: The inputs that break the semantics are known and the control an attacker has over them, but the exploit ability of them remains uncertain. This phase aims to produce a valid exploit for the desired side-channel.

In the case we want to focus the design of the tool for the search of Speculative Control Flow Hijacks, we must specify these phases respectively to previously defined ones.

3.1 Preprocessing: Finding good candidate branches to target for speculative execution

To define the interesting branches we must determine which condition are they based on, therefore, the speculative window will vary depending in the comparison arguments. First, we will consider only the branches leading to a long speculative window (based on the comparison arguments), doing this filtering on all the branches.

We're interested on these denominated *long speculative windows* because the room they provide an attacker to perform an illegal action masked under a misprediction. For example, the attacker can provide via the interface variable data on which the victim will depend to perform loads that later will cause a condition. Because the internal dependencies causing these loads to occur, the attacker can produce some delay in the condition resolution by forcing the load of one of the elements to take more time (e.g. one value is located in a lower memory hierarchy than other). In this situation, the attacker can produce misprediction with an illegal value for the condition but because the other value has not been resolved, the prediction in the wrong branch may happen (depending on the branch predictor state).

An example of short speculation is the comparison against a constant value:

```
void fun(int len){
  int arr[len];
  for (; len > 0; len--) {
    arr[len-1] = len;
  }
}
```

Listing 1: Short speculative window snippet of C code

The resulting LLVM IR explicitly does the comparison against the constant value:

```
for.cond:                                         ; preds = %for.inc, %entry
  %3 = load i32, i32* %length, align 4
  %cmp = icmp sgt i32 %3, 0
  br i1 %cmp, label %for.body, label %for.end
```

Listing 2: Short speculative window snippet of C code

By the other way, a long speculative window requires the size (one of the loaded values) to not be constant in the code, neither cached.

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

Listing 3: Short speculative window snippet of C code

With the next resulting LLVM IR, which performs the comparison the branch depends on of two loaded values:

```
  %0 = load i64, i64* %x.addr, align 8
  %1 = load i32, i32* @array1_size, align 4
  %conv = zext i32 %1 to i64
  %cmp = icmp ult i64 %0, %conv
  br i1 %cmp, label %if.then, label %if.end
```

Listing 4: Short speculative window snippet of C code

We can observe directly in the `cmp` instruction that both arguments come from a memory load. With an optimization pass, still value will come from a load (it doesn't matter).

The purpose of this module is to detect and instrument only the branches which may be interesting. This pass only inserts a very lightweight instrumentation before that branch, keeping the code to be high performant.

3.2 State exploration: finding inputs that reach those branches

In this module we use fuzzing techniques that generate payloads to reach the branches, no misprediction needs to be performed (when the last possible branch is reached, the execution ends).

This is the most sensitive module regarding to what's already offered in matter of fuzzing, as there are very different approaches to perform this task. Also some other payloads generation could be used, like concolic execution, but whose scalability is uncertain for arbitrary programs.

This step may cause false-negative results, as is a problem inherent to fuzzing. When a crash produced by a trampoline is reached, that trampoline will never arise a crash again.

3.3 Determine the attacker controlled values

After obtaining a series of inputs that provoke the program to reach instructions that can be mispredicted. The payloads have to be analyzed in the control and data flow planes to determine if the branch will depend on arbitrary data (one of the loads is attacker controlled). If so, the attacker can really stimulate the branch predictor and provoke a prediction to be wrong, therefore, masking an illegal action during misprediction, that can result in a information leak.

3.4 Filter out cases that cannot lead to Speculative Control Flow Hijack

When we reach this section of the analysis, we have a set of inputs that can lead to a controlled misprediction, but is not clear if they may lead to a Speculative Control Flow Hijack, neither if they can leak arbitrary data. So the purpose is to denote if during the semantically illegal actions performed during a misprediction an attacker can perform a Speculative Control Flow Hijack. A way to lead to a Speculative Control Flow Hijack is by changing the return address in the stack or a branch to a value defined dynamically: either in memory or a register. This step must check factually that the hijack is possible by changing the data controlled values.

The vulnerability looked for during this phase can be other than the Speculative Control Flow Hijack, like an speculative leak. But because the capabilities the Speculative Control Flow Hijack offers, we keep it, although any vulnerability based on the misprediction using controlled values can be detected.

4 Implementation

For each one of the previously defined phases we propose and define a implementation that enables the search of gadgets that can result in a Speculative Control Flow Hijack.

4.1 Semantics accommodation: Compiler pass

To fit to the semantics of speculative execution, we consider the original program and look for the points where speculative execution could be hijacked to perform as an attacker desires.

For this task, we perform an analysis at the compiler level, requiring from the source of the program. At the latest stage of the compilation, before the transformation to machine code, every branch is checked to be dependent in at least two separate memory loads. For those that assert the condition, a trampoline is inserted before the branch is executed, otherwise, they remain the same.

To implement this compiler pass we use LLVM [6], a compiler focused in the modularity of code transformations and which is highly documented. The pass works by producing the Data Dependence Graph of the code analyzed (by functions), and starting from the conditional branch instructions (the ones that will be predicted), following the dependencies between instructions, determines the necessity of introducing the trampoline for a function.

Each trampoline has an unique identifier, used as an argument in the function call.

The function pass the identifier received and its own data structure to another function called `__generic_trampoline__` which determines if the execution should continue or crash should occur. The function called by the trampoline looks like the next in LLVM IR:

```
define void @"trampoline@examples/compiled/base_cases.bc"(i32 %0) {
trampoline_block:
  call void @__generic_trampoline__(i32 %0, [3 x i1]* ↩
       @trigger_examples_compiled_base_cases_bc)
  ret void
}
```

Listing 5: Inserted function called in the trampolines

The global variable is a binary array, to consult and flip the values is designated to the `__generic_trampoline__` function, which is implemented externally by another piece of code linked with the one that receives the pass.

```
@trigger_examples_compiled_base_cases_bc = global [3 x i1] [i1 true, i1 true, i1 true]
```

For projects with a codebase that depends in a complex build system and can insert additional instrumentation to the binary, we do it by setting the compiler to `clang` and the flags `-O3 -flto`

`-save-temps`, which respectively: optimize the resulting code to reduce the memory accesses (but which can be dropped), enables link-time optimizations (and produces object files in LLVM IR format) and save the intermediate LLVM IR bitcode files in the compilation process.

To perform the backwards analysis there are snippets of code that may have a dependency on the control flow outcome, we consider both incomes of that block to look for the load, although they should be considered "weak" critical points because they could not really fill the constraints at their call time.

In the case the global array is set at false, that means all the trampolines will have no effect and the program will run in its regular way.

4.2 State exploration: Fuzzing

The approach selected to obtain an attacker controlled inputs that reach the critical points is fuzzing. The fuzzing driver already states the input parameters to a program snippet, so the input. In the fuzzing process, the fuzzer generates inputs for the program to test with a high-throughput. Each one of the generated inputs is converted into a valid payload to be analyzed later in the case it reaches a critical point never reached before.

4.3 Taint analysis: Data flow analysis

With the payloads produced by the fuzzing process, it needs to be determined the relation between the payload and the control flow, to reach that branch. After finding that correlation, the data not inferring in that decision can be treated as symbolic, for which the static analysis will deduce if the Speculative Control Hijack is possible.

For this task we make use of a tool that uses as input the program, the reached branch and the payload generated. The tool performs a taint analysis on the generated payload and determines which parts of it can be controlled by an attacker. The outcome should tell if an attacker can still make the condition variable on the input and my modifying which parts of the input to do it.

4.4 State evaluation: Dynamic analysis

The processor is agnostic about if mispredictions and information leaks can be harmful, and all this information must be ensured from the software side.

The information regarding the potential an attacker has during misprediction has to be checked against the legal semantics of the code during misprediction. Considering too the Out-of-Order execution, so instructions not dependent on ones that produce an exception can be executed as well.

For this purpose we may consider special semantics for each one of the instructions during misprediction, ignoring a faulty execution. The instructions to be checked are only the ones falling in the speculative window. For this task we make use of a custom instruction interpreter used during the speculative out of order execution and takes as input for execution the input with symbolic regions.

5 Evaluation

To demonstrate the validity of the proposed model we run it against OpenSSH v8.0 [7]. We select this tool because it should have the characteristics side-channel resistant tool have to present against our threat model: A well defined program running in a trusted environment and that offers points that lead to code of specific parts of its codebase.

After running the compiler pass over the generated files, it inserts a total of **1247** calls to trampolines among the **20577** conditional branches.

As shown in Table 1 there's not a direct correlation between the number of conditional branches a program presents and the inserted trampolines. This is mostly because the layout of the code: files with heavy checks against static parameters (e.g. configuration parsers) are more prone to have a huge number of conditional branches that cannot be tweaked to perform speculation. There are files for which the trampolines are not inserted because of a fail in the Data Dependence Graph generation; due this fact, we choose an modular and incremental trampolines insertion, before the linkage procedure to avoid that errors can affect the trampolines insertion of the rest of the codebase.

Although the attack surface has been reduced, it still cannot be ensured if any of those branches can lead to a Speculative Control Flow Hijack. Anyway, it reduces drastically the branches to analyze to a

File	**B**	**T**	File	**B**	**T**	File	**B**	**T**
log.o	63	0	sandbox-darwin.o	1	0	monitor.o	314	35
platform-misc.o	1	0	packet.o	499	40	sshbuf-misc.o	27	**X**
auth2-passwd.o	9	0	sshkey-xmss.o	1	0	sshpty.o	23	4
ssh-pkcs11-helper.o	74	11	gss-serv.o	1	0	gss-serv-krb5.o	1	0
kexgex.o	23	0	utf8.o	69	7	rijndael.o	33	0
authfd.o	149	3	match.o	100	**X**	ge25519.o	14	0
nchan.o	76	1	auth2-none.o	5	1	kexgexc.o	33	6
sandbox-solaris.o	1	0	kexgen.o	62	8	ttymodes.o	391	1
sandbox-pledge.o	1	0	openbsd-compat/vis.o	97	6	openbsd-compat/bsd-statvfs.o	1	0
openbsd-compat/port-solaris.o	1	0	openbsd-compat/fake-rfc2553.o	1	0	openbsd-compat/strcasestr.o	1	0
openbsd-compat/bsd-flock.o	1	0	openbsd-compat/rresvport.o	1	0	openbsd-compat/bindresvport.o	13	0
openbsd-compat/bsd-malloc.o	1	0	openbsd-compat/bsd-misc.o	2	0	openbsd-compat/rmd160.o	1	0
openbsd-compat/bsd-getline.o	1	0	openbsd-compat/strtonum.o	14	0	openbsd-compat/getrrsetbyname.o	74	10
openbsd-compat/port-irix.o	1	0	openbsd-compat/bsd-setres_id.o	1	0	openbsd-compat/strptime.o	1	0
openbsd-compat/mktemp.o	28	0	openbsd-compat/basename.o	1	0	openbsd-compat/strlcat.o	8	0
openbsd-compat/bsd-openpty.o	1	0	openbsd-compat/bsd-closefrom.o	14	0	openbsd-compat/bsd-nextstep.o	1	0
openbsd-compat/blowfish.o	76	**X**	openbsd-compat/realpath.o	39	1	openbsd-compat/sha1.o	1	0
openbsd-compat/inet_ntoa.o	1	0	openbsd-compat/openssl-compat.o	5	0	openbsd-compat/bsd-waitpid.o	1	0
openbsd-compat/strndup.o	1	0	openbsd-compat/strtoul.o	1	0	openbsd-compat/port-net.o	17	0
openbsd-compat/port-aix.o	1	0	openbsd-compat/recallocarray.o	13	0	openbsd-compat/readpassphrase.o	36	0
openbsd-compat/libressl-api-compat.o	13	0	openbsd-compat/reallocarray.o	1	0	openbsd-compat/explicit_bzero.o	1	0
openbsd-compat/port-uw.o	1	0	openbsd-compat/timingsafe_bcmp.o	9	0	openbsd-compat/bsd-signal.o	5	0
openbsd-compat/strtoull.o	1	0	openbsd-compat/bsd-cygwin_util.o	1	0	openbsd-compat/getopt_long.o	72	18
openbsd-compat/bcrypt_pbkdf.o	21	0	openbsd-compat/getcwd.o	1	0	openbsd-compat/strlcpy.o	5	**X**
openbsd-compat/strnlen.o	1	0	openbsd-compat/bsd-poll.o	1	0	openbsd-compat/bsd-err.o	1	0
openbsd-compat/kludge-fd_set.o	1	0	openbsd-compat/arc4random.o	41	3	openbsd-compat/sha2.o	1	0
openbsd-compat/bsd-getpeereid.o	3	0	openbsd-compat/freezero.o	2	0	openbsd-compat/strtoll.o	1	0
openbsd-compat/fmt_scaled.o	86	20	openbsd-compat/sigact.o	1	0	openbsd-compat/setenv.o	1	0
openbsd-compat/setproctitle.o	34	3	openbsd-compat/daemon.o	1	0	openbsd-compat/strsep.o	1	0
openbsd-compat/inet_ntop.o	1	0	openbsd-compat/getgrouplist.o	1	0	openbsd-compat/base64.o	51	**X**
openbsd-compat/bsd-snprintf.o	1	0	openbsd-compat/xcrypt.o	10	1	openbsd-compat/bsd-asprintf.o	1	0
openbsd-compat/dirname.o	1	0	openbsd-compat/strmode.o	11	0	openbsd-compat/getrrsetbyname-ldns.o	1	0
openbsd-compat/glob.o	269	42	openbsd-compat/bsd-getpagesize.o	1	0	openbsd-compat/inet_aton.o	1	0
openbsd-compat/pwcache.o	15	0	openbsd-compat/port-linux.o	60	1	openbsd-compat/md5.o	1	0
sandbox-null.o	1	0	monitor_fdpass.o	14	1	xmss_wots.o	1	0
clientloop.o	481	68	cipher-chachapoly.o	9	0	xmss_commons.o	1	0
umac.o	69	7	kexecdh.o	23	0	sandbox-capsicum.o	1	0
moduli.o	115	5	ssh-keysign.o	68	7	serverloop.o	192	19
sftp-server-main.o	2	0	loginrec.o	70	2	platform-pledge.o	1	0
groupaccess.o	26	1	digest-openssl.o	33	1	ssh-xmss.o	1	0
ssh-rsa.o	82	7	auth2-hostbased.o	46	10	auth2-chall.o	71	7
dh.o	95	9	ssh-pkcs11-client.o	71	5	auth-shadow.o	14	1
entropy.o	4	0	ssh-agent.o	264	24	audit-bsm.o	1	0
auth-rhosts.o	59	3	ssherr.o	86	0	cipher-ctr.o	1	0
sandbox-rlimit.o	1	0	auth-krb5.o	1	0	sftp-glob.o	8	0
gss-genr.o	1	0	cipher-aes.o	1	0	monitor_wrap.o	204	3
msg.o	10	1	readpass.o	34	0	audit-linux.o	1	0
cleanup.o	1	0	sshbuf-getput-crypto.o	28	2	sandbox-systrace.o	1	0
dispatch.o	45	3	xmss_hash_address.o	1	0	sshbuf-getput-basic.o	126	5
auth.o	171	18	sftp.o	594	27	session.o	407	**X**
bitmap.o	67	21	xmss_hash.o	1	0	krl.o	626	130
readconf.o	823	**X**	channels.o	1194	**X**	hostfile.o	153	10
sshkey.o	1147	**X**	scp.o	435	**X**	sshconnect2.o	381	32
fatal.o	2	0	cipher.o	88	2	hash.o	2	0
addrmatch.o	111	24	xmss_fast.o	1	0	auth2.o	186	14
fe25519.o	44	31	ssh.o	483	47	auth2-kbdint.o	6	0
crc32.o	5	0	authfile.o	129	0	auth2-gss.o	1	0
ssh-keygen.o	680	**X**	poly1305.o	25	0	hmac.o	22	6
canohost.o	28	0	auth-options.o	294	**X**	kexgexs.o	33	6
sc25519.o	133	**X**	auth-sia.o	1	0	verify.o	1	0
auth2-pubkey.o	187	28	sshlogin.o	8	0	ed25519.o	56	0
kexsntrup4591761x25519.o	21	0	ssh-ecdsa.o	41	1	ssh-ed25519.o	35	1
ssh-add.o	134	16	md5crypt.o	1	0	auth-passwd.o	21	1
sftp-common.o	43	10	ssh-keyscan.o	162	**X**	uidswap.o	34	2
sandbox-seccomp-filter.o	8	0	digest-libc.o	1	0	sshd.o	455	45
platform.o	2	0	sftp-client.o	392	25	smult_curve25519_ref.o	520	0
sftp-server.o	823	65	sntrup4591761.o	209	**X**	cipher-aesctr.o	1	0
sshtty.o	9	0	auth-bsdauth.o	1	0	audit.o	1	0
ssh_api.o	102	3	umac128.o	73	7	kexc25519.o	21	0
ssh-dss.o	43	0	ssh-pkcs11.o	300	37	chacha.o	32	0
uuencode.o	9	**X**	platform-tracing.o	3	0	kexdh.o	40	0
sshconnect.o	268	34	mux.o	571	40	misc.o	547	**X**
auth-pam.o	1	0	servconf.o	876	37	sshbuf.o	200	86
mac.o	55	0	compat.o	47	1	kex.o	279	16
atomicio.o	51	2	xmalloc.o	15	0	dns.o	55	2
progressmeter.o	47	7						

Table 1: Number of conditional branches and trampolines inserted in OpenSSH v8.0 for each file. **B** stands for conditional branches and **T** for trampolines. **X** represents a Data Dependence Graph failed generation

6%, it avoids additional checks and useless branch prediction emulation (which although we're not using in our analysis, it may be used).

We can observe in this library prefiltering results some characteristics that confirm the oblivious code design for cryptographic libraries. For example, this lack of differential observability is present on `smult_curve25519_ref.o`, a implementation to handle a specific elliptic curve; although it contains a huge number of conditional branches (**520**), none of them depends on two or more load instructions, which makes sense because for code that may have an oblivious layout.

To improve the fuzzing throughput, *libFuzzer* allow a parallel execution, and additionally, as a file contents are modified, we implement the backup of the data to be concurrent. This is done by a lock and release at backup time, as the backups dpend on a single byte write.

OpenSSH provides a set of fuzzing drivers for its exposed interfaces (`authopt_fuzz.cc`, `privkey_fuzz.cc`, `kex_fuzz.cc`, `pubkey_fuzz.cc`, `sig_fuzz.cc`, `pubkey_fuzz.o` and `sshsig_fuzz.cc`, `sshsigopt_fuzz.cc`) and one common to all of them (`ssh-sk-null.cc`). We adapt the common file in the fuzzing driver to implement the generic trampoline, that at its time will depend on each one of the fuzzing drivers to reference the global variables that should be triggered.

At the beginning of the fuzzing process, if the backup file is not created, it creates it and maps the internal global variables with the data know if the trampolines should be triggered (based on the previous executions that have persisted the global variables in the backup file). The generic trampoline receives the index to be consulted for a specific global variable and if it should crash, before it does to backup the state.

```
void __generic_trampoline__(uint32_t index, uint8_t *to_flip) {
  if (to_flip[index]) {
    __backup_data__(index, to_flip);
    abort();
  }
  return;
}
```

Listing 6: Generic trampoline implementation in C code

For building the trampolines we use the build system available at build.sh at OSS-Fuzz/OpenSSH, and we use the corpus available at openssh-fuzz-cases.

Among the different fuzzing runs tried to demonstrate the functioning of the tooling we get payloads corresponding to specific trampolines.

For example, the binary file `sshkey_private_deserialization` is the result of linking 46 different object files (out of 280). It contains 302 trampolines, and it's fuzzing after 1 minute can trigger 51 of them by 32 payloads, but all of them result from the `sshbuf.o` instrumentation. The fuzzing of the binary file `sshsig_fuzz` (which links the same 46 files) during 1 minute, has a broader set of sources of the trampolines triggered apart from `sshbuf.o` such as `arc4random.o` or `ssh-rsa.o`.

To determine if the fuzzing process is unable to reach more trampolines or that they're directly not reachable from the entry point will depend on the fuzzing driver and the control flow of the program that may not able to reach in any case a subset of the trampolines. It either can happen that there are parts of the code not reached because the code that comes before has a high complexity.

6 Future work

This project aims to detect the Speculative Control-Flow Hijack produced in a given codebase and which an attacker could use to gain execution privileges. Given that it is a project which uses the identification of possible branches that can be violated in the compilation phase, that same instrumentation can be used to harden those branches demonstrated to produce a Speculative Control-Flow Hijack in the later stages of the analysis. Mostly, this project improvements come from the side of having a proper taint analysis and later static analysis. The tooling for the taint analysis should be able to determine automatically the attacker-controlled parts of the input. The static analysis phase must be crafted with the interpretation of the microarchitecture semantics, as related work has done.

We believe that this work settles a ground for a further performance-aware side-channels protection. The methodology we present can be applied as well to other type of attacks.

7 Related Work

Prior work in the detection of side-channel exploitation of Spectre-based side-channel attacks aim to detect the point where misprediction may happen in order to leak a secret. Whereas, the concept of secret and how an attacker is able to perform a leak is not defined in these models. Although these projects can detect information leaks, they don't state how to do it and if this would be harmful. They can have several an excess of false positives mostly because the threat model is very relaxed and the device constraints to present side channels are not considered (in our model this is done at the last step, the static evaluation, and totally independent from all the previous analysis).

In the static-analysis field detection there are pattern-matching based approaches [8] run through the code either at the source or the binary looking for patterns similar to a basic Spectre v1. The analysis is simple and performed in a naive way, intermediate instructions or different patterns with the same result, may do the analysis to fail.

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

Listing 7: Spectre V1 in C code

```
# %bb.0:
    movl    array1_size(%rip), %eax
    cmpq    %rdi, %rax
    jbe .LBB0_2
# %bb.1:
    leaq    array1(%rip), %rax
    movzbl  (%rdi,%rax), %eax
    shlq    $9, %rax
    leaq    array2(%rip), %rcx
    movb    (%rax,%rcx), %al
    andb    %al, temp(%rip)
.LBB0_2:
    retq
```

Listing 8: Spectre V1 in assembly code

Spectector [9] defines *speculative-non-interference*, a mathematical definition of when a program leaks information under speculative execution; this concept needs to be checked over the critical parts of the program. To demonstrate it's functionality, it constructs a tool that produces all the program traces under concolic execution and checks if any of them breaks *speculative non-interference*. Although the methodology is strong, it depends on static analysis and has an explosive complexity. Also, the tool implements its own semantics of the instructions, making the project to work in controlled cases, but leaving lot of undefined instructions which leaves several programs.

In the same field of dynamic-finding vulnerabilities, it exists oo7 [10], a tool that performs speculative execution too

SpecFuzz [11] instruments LLVM compiler to insert at code generation time a soft speculation, executing the semantically wrong branch for the next Basic Block and later returning to the correct outcome of the branch. The basic block corresponding to the wrong prediction is compiled with sanitizers, so every illegal access will produce a soft crash (the sanitizers are modified fro that), and then continue the execution from the right branch. This model of speculative execution is incomplete, if a crash happens in the code, later ones won't be reported, additionally, it doesn't check if the attacker can force the misprediction of the branch. While embedding the speculative execution in the binary produced and getting executed, the faults produced may not correspond to real threats of the program, and to harden the binaries based on the provided information of simple crashes without making clear if they can really happen will lead to high false-positives. At the same time there will be missing false negatives because the model of speculative execution they embed in the executable.

SpecTaint [12] is the most recent work in this area, it embeds speculative execution semantics into the *qemu* emulator, and generate seed inputs using classic fuzzing techniques (AFL and honggfuzz). With a similar fashion as SpecFuzz, the state of the processor execution is saved before the misprediction begins and it's restored after it ends, but it reports all the possible crashes until the speculation ends.
In the proposal we make, the attacker capabilities are inherent to the process, so, first with the prefiltering of the branches that can be hijacked, we facilitate both the state exploration and state evaluation by leaving only branches that could be hijacked, and the taint analysis finally provides that information. The

method is incremental and because its modularity, can be customized to other type of Spectre variants and side-channel attacks.

7.1 Problematic of evaluation

To evaluate the effectiveness of results that a tool provides, it's necessary to have a methodology for it, but the set of tools that exist at the current moment to detect Speculative type attacks has a lot of variety and doesn't focus into the same objective. Currently, under our knowledge, there are not tools that can demonstrate the possibility of an attack which has its source in an speculative execution attack. In the other sense, these tools can provide points where an arbitrary speculative leak can happen, but the capabilities of the attacker to perform that leak get blurred for each program. Also, it's not possible to give a generalization for how the search may be done because the model threats may vary, and giving different capabilities to the attacker can result in checks with the wrong resolution.

Additionally, the microarchitectural model of the processors is often closed and obscure, therefore, it's hard to assert the instructions microarchitectural behavior in all the microarchitecture, but small snippets of this behavior may be inferred.

8 Conclusion

Side-channel attacks are here to stay, the methodology for vulnerabilities produced by them has to receive more attention to keep the systems as secure as possible. Because the complexity of nowadays systems, the difficulty to asses their security grows with their size. Still, the complexity to find a vulnerability increases as more are patched, but that doesn't mean they stop to exist.

If we want to preserve the existing software in the systems to be assessed as secure with a high confidency, we need to ensure their protected even against side-channel attacks. In the same way it cannot be ensured security at other levels of abstraction of the system, side-channels remain as a problem which needs from its own security analysis. All the similarities between the side-channel attacks and the classical vulnerabilities can be neared too in the analysis of vulnerabilities too, because their nature is similar; furthermore, two different layers of vulnerabilities can be combined to perform an attack.

References

[1] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *2013 IEEE Symposium on Security and Privacy*, pages 191–205. IEEE, 2013.

[2] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[3] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 279–299. Springer, 2019.

[4] Hardware Sanitization.

[5] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. Sok: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[6] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[7] OpenSSH project.

[8] Spectre variant 1 scanning tool, 2018.

[9] Marco Guarnieri, Boris Köpf, Jose F. Morales, Jan Reineke, and Andres Sanchez. Spectector: Principled detection of speculative information flows. In *IEEE Symposium on Security and Privacy*. IEEE, May 2020.

[10] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019.

[11] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[12] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. 2020.