A Software Mitigation Approach for Branch Target Injection Attack

Ke Sun, Kekai Hu, Henrique Kawakami, Marion Marschalek, Rodrigo Branco (BSDaemon)

Intel - STrategic Offensive Research & Mitigations (STORM)

v1.42 – https://github.com/intelstormteam/Projects/randpoline

0. OBJECTIVES

During our work in battling various speculative-side-channel vulnerabilities, which involved writing proof-of-concept exploits, new attack/threat identification, mitigation design/evaluation, and many other tasks, many ideas for mitigations have been discussed (some end up in real implementations, some are disclosed as patents or published in other forms). This whitepaper shares one of those ideas that our team considered, including the theory and possible implementation options, together with an example implementation in the form of a GCC plugin, with the objective to give a base for other researchers to improve upon and also for the industry to consider.

1. BACKGROUND

Speculative execution is a technique used by most modern processors to achieve high performance, which allows instructions to be executed a priori of knowing that they will be architecturally required. An important part of speculative execution involves branch prediction: instead of waiting for the target of a branch instruction to be resolved, the processor looks ahead to predict the control flow and speculatively executes the instructions on the predicted path.

However, despite the performance improvement, it is known to the public that speculative execution can be leveraged by a malicious attacker to bypass security boundaries due to the following behaviors:

1. Speculative execution may not necessarily be bounded by architectural security checks (e.g. memory range, privilege level, etc.)

2. Cache lines filled by speculative execution are not reversed when the speculatively executed code is dropped, thus leaving measurable trace to architectural execution

3. Branch prediction resources (e.g. source and target addresses) are shared by entities with different security contexts (such as user mode code and kernel mode code, different processes on the same Hyperthread, different Hyperthreads of the same physical core, etc.)

These conditions led to vulnerabilities publicly known as Spectre and Meltdown, impacting various architectures and platforms, including Intel's x86 processors.

More specifically, the branch target injection attack of Spectre, also known by Google Project Zero's definition as Issue Variant 2, takes advantage of the indirect branch predictors inside the processor: a malicious attacker can influence the branch prediction and thus the speculative execution of the victim process by injecting a speculative branch target into the indirect branch predictor. The speculatively executed code in the victim process can carry out secret-dependent cache loading that is measurable to the attacker and eventually leaks the secret value.

This paper describes a software-based approach to help mitigate branch target injection side-channel attacks.

Currently the existing mitigations include both hardware- and software-based solutions. An example of a hardware-based solution is Intel’s new interface between the processor and system software:

[Quoting from “Intel Analysis of Speculative Execution Side Channels” white paper]

“This interface provides mechanisms that allow system software to prevent an attacker from controlling the victim’s indirect branch predictions, such as flushing the indirect branch predictors at the appropriate time to mitigate such attacks.”

"There are three new capabilities that will now be supported for this mitigation strategy. In particular, the capabilities are:

• Indirect Branch Restricted Speculation (IBRS): Restricts speculation of indirect branches.

• Single Thread Indirect Branch Predictors (STIBP): Prevents indirect branch predictions from being controlled by the sibling Hyperthread.

• Indirect Branch Predictor Barrier (IBPB): Ensures that earlier code's behavior does not control later indirect branch predictions."

For the software-based solutions, the most widely known one is called "return trampoline" or "retpoline", which is a compiler-supported mitigation technique that converts all vulnerable indirect branches into (1) push *[target address]* and (2) ret. For example, with retpoline, an indirect jump will be translated into: (1) a dummy call that pushes a dummy and safe return address on both the process stack and the return stack buffer (RSB); (2) overwrite the return address on process stack to the indirect branch target; (3) ret to the target address.

In this case, the indirect branch is carried out in the form of a return instruction, the speculative execution of which is controlled by the top RSB entry, which is already populated with a safe location.

2. DESCRIPTION OF THE APPROACH

In this paper, a different software-based idea is introduced to help mitigate a branch target injection type of side-channel attack, as an alternative to retpoline. The basic theory is to break the deterministic conditions of such attack by randomizing both the branch history and also the "from address" of the vulnerable indirect branch. This approach is therefore named "random trampoline" or "randpoline".

In order for a branch target injection attack to succeed, the attacker needs to inject one or more branch prediction entries that will be speculatively consumed by the victim process. In order for

this to happen, the attacker needs to run the “training code” which satisfies the following conditions:

1. Contains an indirect branch with the same (or aliased) linear address as the vulnerable indirect branch inside the victim process

2. In some conditions, the branch history immediately before the indirect branch also has to match between the attacker and the victim

Therefore if randomization is introduced to one or both of these two factors, the possibility of successful branch target injection attack can be dramatically reduced to the level that it is unlikely it is exploitable in practice.

Such randomization can be achieved by introducing a randpoline area in memory consisting of continuous indirect branch instructions, (e.g. “jmp rax”), which serves as an intermediate trampoline between the original “from address” and “target address” of a vulnerable indirect branch.

With randpoline, a vulnerable indirect branch can be converted into:

1. A low-latency jump from the original “from address” to the randpoline area with a random offset (no memory access, thus it shall not have enough latency needed for branch target injection attack)

2. An indirect jump from the randpoline area to the original “target address” (has a dependency on memory access and thus is considered vulnerable to branch target injection attacks)

This code pattern randomizes both the “from address” of the vulnerable indirect branch, and also the branch history immediately before it. The random offset should come from a random number

generated at runtime. Without the knowledge of the random offset, the attacker will not be able to deterministically inject a speculative indirect branch entry that matches the victim's execution flow without brute forcing.

3. IMPLEMENTATION GUIDELINE

In general, the implementation of randpoline includes two parts:

(1) Creating a memory area ("randpoline area") consisting of continuous indirect branch instructions, which serves as an intermediate trampoline between the original "from address" and "target address" of a vulnerable indirect branch.

An example of randpoline area is shown below. The "ud2" instruction added after each jmp is used to provide extra robustness against brute force attacks that try to leverage the speculative window.

```
//randpoline area:
0:  ff e3                   jmp    rbx
2:  0f 0b                   ud2
4:  ff e3                   jmp    rbx
6:  0f 0b                   ud2
8:  ff e3                   jmp    rbx
a:  0f 0b                   ud2
c:  ff e3                   jmp    rbx
e:  0f 0b                   ud2

…
```

(2) Convert each vulnerable indirect branch into (a) a low-latency jump from the original "from address" to the randpoline area with a random offset (no memory access dependency) and (b) an indirect jump from randpoline area to the original "target address", as shown in Figure.1

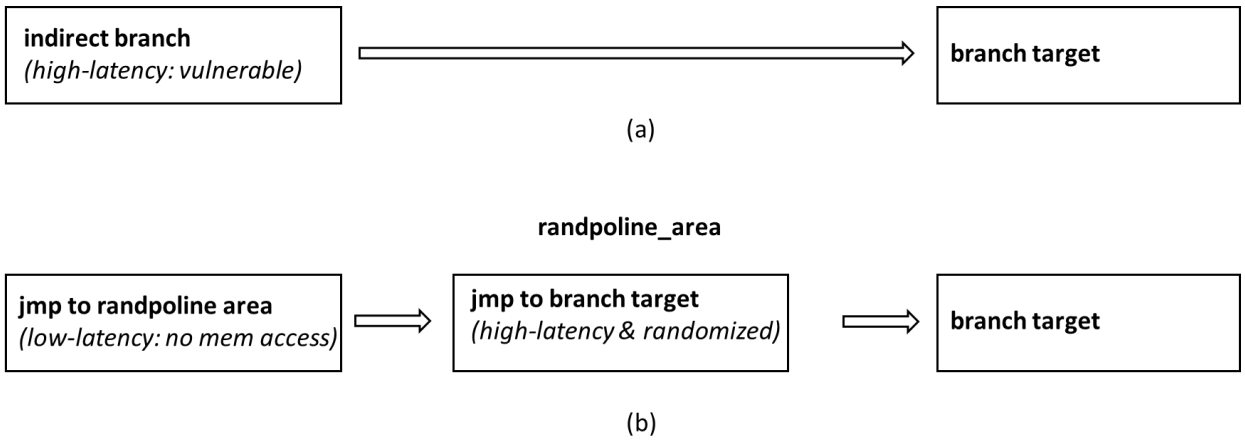

Figure.1 Schematic of indirect branch for (a) unmitigated cases (b) mitigated-by-randpoline case

3.1 EXAMPLE IMPLEMENTATION OF INDIRECT JUMP

To be more specific, Figure.2 gives an example of the randpoline code of an indirect jump. A vulnerable indirect jump such as

```
jmp qword ptr [memory_address]
```

can be converted to the following code under randpoline:

```
lea rax, [randpoline_base + rand_offset * 4]
…
mov rbx, qword ptr [memory_address]
jmp rax
```

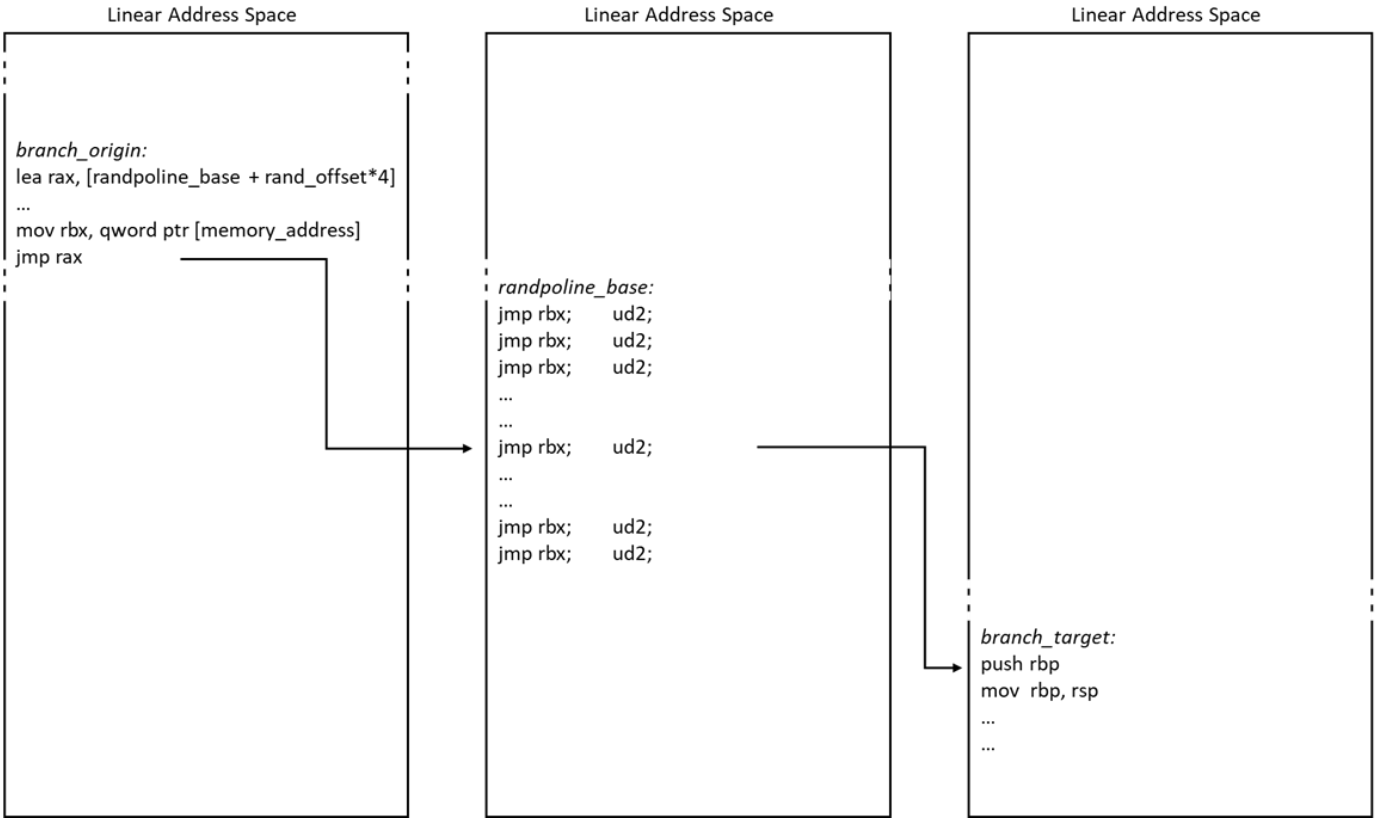

Figure.2 Schematic of randpoline implementation example

where *randpoline_base* is a constant known at compile time or load time, and is used as an immediate, while *rand_offset* is based on a random number generated at runtime, and stored in a register to avoid memory access.

While the jump to randpoline is low-latency in nature due to its lack of memory access dependency, for extra level of robustness, it is desirable to have the target address calculation ahead and distant from the branch itself to further remove any possible latency in corner cases.

The random number can be either a pseudo random number or a cryptographic-strength random number (e.g. from rdrand), and it can be updated per process, per routine, per branch or with

other flexible options (e.g. every N times of indirect branches executed), all depending on the mitigation requirement.

3.2 IMPLEMENTATION OF INDIRECT CALL

A similar mitigation can also be applied to vulnerable indirect calls, such as

```
call qword ptr [memory_address]
```

can be converted to the following code under randpoline:

```
lea rax, [randpoline_base + rand_offset * 4]
…
mov rbx, qword ptr [memory_address]
call rax
```

The callee routine will return to *the instruction following the call*, consistent with the original control flow. It is also possible to implement a multiple-level randpoline which can largely increase the randomization entropy with very limited increase in memory footprint.

4. COMPARISON WITH RETPOLINE

Randpoline can be considered as an alternative software solution to retpoline to mitigate branch target injection attacks. While retpoline provides a deterministic mitigation by forcing mis-prediction on all indirect branches, randpoline uses randomization to provide probabilistic mitigation. Different from retpoline which converts all indirect branches in the form of return, randpoline does not change the nature of the indirect branches and does not affect the matching of call/ret pairs. Therefore it will not cause false alarms in many ROP detection approaches (such as K-bouncer, rsb-miss event profiling, etc.) and is CET compatible. Plus it shall require less engineering effort for integration with other existing control flow mitigation technologies like CFG and RAP (Ref[3]).

While randpoline is a stand-alone software approach to protect the indirect jumps/calls in its scope, a more complete mitigation of branch target injection attack needs to also protect the native return instructions in the code base. Therefore randpoline, in this case similar to retpoline, also requires the support from OS and, on certain platforms, the microcode patch to make sure the return instruction does not consume the branch target array in speculative execution.

5. SECURITY CONSIDERATIONS and LIMITATIONS

Since randpoline is a probabilistic mitigation that shall make the branch injection attack much harder (in practice) by removing the deterministic factor out of an attack, it could be compared to the role of ASLR in the space of branch prediction. Similar to ASLR though, the robustness can be greatly improved if detection mechanisms for continuous attack exist, but they are out of the scope of this discussion. We have considered its effectiveness and limitations regarding the following aspects.

5.1 RANDOMIZATION ENTROPY

The level of security provided by randpoline depends on its effective randomization entropy, which involves the following factors:

- 1. Size and number of randpoline areas: the approximate security entropy (represented in bits, the same below) shall be roughly proportional to $N \times log_2(R)$, where R = number of jumps in each randpoline area, and N = number of chained randpoline areas.
- 2. The total entropy of the Branch Target Buffer (BTB) tagging space (E). Normally BTB uses aliasing instead of the full branch address for tagging. E is micro-architectural and platform specific. Randpoline cannot provide more entropy than E, so its effective robustness is upper-bounded by MIN($N \times log_2(R)$, E). If a platform has a very small E, randpoline may not be able to provide enough security entropy against brute force attack.

The required implementation for a given application should consider how many times an attacker would be able to repeat the attack, and how much information could be leaked by each successful attack.

The re-randomization of randpoline offsets for different branches in the victim application may seem to reduce the security entropy by a factor of N_r, where N_r is the times the randpoline offset gets updated in an attack, since it increases the overall chance for one branch to hit the injected target. This is not true because in a branch target injection attack, the attacker needs to have knowledge or control of the context of the speculative execution in the victim, in other words, it needs to know which branch inside the victim is speculating to the target it injected in order to leverage the behavior. An unknown branch hitting the injected target does not in general help the attack and may, on the opposite, make it harder by adding noise to the possible output.

In corner cases where there are many closely-spaced branches taking place with the same or similar context, it should be avoided to use the per-branch re-randomization and use re-randomization at higher level (e.g. per routine) for the randpoline offsets.

5.2 BRUTE FORCE ATTACK

It might be assumed that a simple brute force attack can be carried out by (1) injecting the target for all possible branches to hit the victim branch, or (2) triggering a victim branch many times to hit the injected target(s), or both. However, there is one key fact that randpoline is leveraging which makes brute force attacks more difficult and non-deterministic: **BTB is a cache-like array with limited number of entries which are dynamically updated by all the branches all the time.**

1. The number of BTB entries is much smaller (usually orders of magnitude) than the total tagging space of branches. **Therefore it is not possible to inject targets for all possible branches.** If the attacker tries to do so, it will overwrite the entries it injected and still end up with a very small probability coverage. The same scenario also applies to brute forcing the victim branch many times: due to the limit number of BTB entries, the injected target(s) will be purged way before the victim branch has a good chance to hit it.
2. Secondly, the BTB is updated by all the branches all the time, including the flow of the attacker, flow of the victim, context switches, interrupts, etc. Thus, it is not even possible to pollute all the BTB entries.

Therefore, although the attack scenarios mentioned above may increase the brute force coverage in one attack, the attacker will still need to iterate and accumulate statistically enough attempts for one successful hit, and repeat the same effort for each byte that is leaking due to the branch randomization.

Considering there are always noises in cache based side channel measurements plus the noise introduced by randpoline itself (as mentioned in previous section), additional levels of statistical complexity will be required to resolve real signal from noise.

In summary, brute force attack against randpoline is in theory possible but expected to be very difficult in practice. Besides, if a more robust brute force prevention is desired, the implementation can always require a hardware flush of the BTB using IBPB every M times of repetition, where $1 << M << 2^{MIN(N \times log2(R), E)}$ /(number of BTB slots injected by attacker per iteration), which shall add negligible amount of performance impact for most values of M in practice.

5.3 DATA-ONLY SELF-TRAINING ATTACK

While the primary scope of randpoline is cross-domain branch target injection attack, the risk of data-only self-training attack shall also be reviewed. For a per branch randomizing implementation, such attack shall not be a concern. For randpoline with re-randomization at higher level, there can be chance for self-training attack to happen between adjacent branches within the same randomization window. One solution to such risk is to add a code-based re-randomization: triggering the randpoline offset update (which has very low overhead) after the code pattern of a memory leaking gadget, given it is feasible to identify such gadget at compile time.

5.4 CONVENTIONAL CACHE-BASED SIDE CHANNEL ATTACK

Another possible attack on randpoline is to use the conventional cache-based side channel attack combined with branch target injection attack: firstly, using the conventional cache-based side channel attack (e.g. “PRIME + PROBE”) to probe the randpoline offset (it still cannot resolve the within cache-line offset, but can get close enough for the attacker to brute force); secondly, with the knowledge of the randpoline offset, carrying out the branch target injection attack.

This attack scenario is only possible when the randpoline implementation uses a very infrequent re-randomization of the branch offset and the difficulty of such attack also depends on the noise level caused by the cache activities in the system and specific implementation details (Ref[2]). In practice, it is not easy to carry out conventional cache set collision attack in a non-ideal environment (in theory any secret-dependent cache loading is vulnerable to set collision attack, yet the actual cases for successful attack are much more narrowed).

Therefore, even for the randpoline implementation that is vulnerable to conventional cache-based side channel attack, requiring such an attack as prerequisite itself is making the branch target injection attack a lot more difficult.

5.5 LATENCY

It is a common understanding that in branch target injection attack, the latency in resolving the branch target is a necessary primitive that creates enough window for speculative execution. As mentioned in the original publication for Variant 2 (later named Spectre) (Ref[1]), non-cached memory access latency of the branch target is needed for a successful attack, which is consistent with the observation in our tests that a low-latency branch with no dependency on memory access does not give enough window for speculative execution of a memory leaking gadget, even in a port contention scenario. Therefore the low latency branch to the randpoline area is in general considered not vulnerable to branch target injection attack.

However, it is hard to absolutely rule out any potential corner case that could break such assumption (e.g. maybe with some extreme processor resource contention). It is also possible that an attacker may try to race with the low latency branch using a shorter gadget; for example, instead of leaking the memory as in a V2 attack, it leaks the existing value in a register. Assuming this is possible in certain conditions (maybe even by using other side-channels), it might be further leveraged to attack randpoline, given that randpoline keeps the random offset in a register. Counter measures include using “per code entry” re-randomization, dedicating register usage to eliminating possible gadget, and possibly others. Another hardening to address the potential risk of a latency race is to use direct branch into randpoline area with immediate offset that get dynamically patched at runtime.

In summary, randpoline is a general idea for the security community to consider, review and improve upon. While its effectiveness has been demonstrated in the tests we conducted, it has not been comprehensively tested against all possible corner cases and its security effectiveness may vary according to the actual implementation and the platform-specific micro-architecture.

6. PERFORMANCE CONSIDERATIONS

While it is not a comprehensive performance evaluation, the following results are observed with a prototype implementation using a GCC plugin (available as GPLv2 on github):

Test Environment	Runs	Unpatched total ms	Randpoline total ms	Retpoline total ms	Randpoline over unpatched (time increase)	Retpoline over unpatched (time increase)	Randpoline over retpoline (time decrease)
GCC6 gcc version 6.5.0 (Debian 6.5.0-1)	*Run1*	62736	64383	74915	2.63%	19.41%	14.06%
	Run2	62901	64790	74740	3.00%	18.82%	13.31%
	Run3	63099	64164	74857	1.69%	18.63%	14.28%
	Run4	63167	64366	74958	1.90%	18.67%	14.13%
	Run5	63221	64556	75129	2.11%	18.84%	14.07%
	Run6	62907	64420	75564	2.41%	20.12%	14.75%
	Run7	62699	64714	75915	3.21%	21.08%	14.75%
	Run8	63153	64341	75279	1.88%	19.20%	14.53%
	Run9	62871	64624	75453	2.79%	20.01%	14.35%
	Run10	62878	65278	74933	3.82%	19.17%	12.88%
	Average	**62963.2**	**64563.6**	**75174.3**	**2.54%**	**19.40%**	**14.11%**
	StdDeviation	**184.5925242**	**314.371401**	**373.0853492**	**0.68%**	**0.79%**	**0.60%**
GCC7 gcc version 7.3.0 (Debian 7.3.0-30)	*Run1*	63141	64053	74751	1.44%	18.39%	14.31%
	Run2	62866	64470	75010	2.55%	19.32%	14.05%
	Run3	63289	64137	75364	1.34%	19.08%	14.90%
	Run4	62746	63968	75473	1.95%	20.28%	15.24%
	Run5	62576	64224	75235	2.63%	20.23%	14.64%
	Run6	62904	64283	74658	2.19%	18.69%	13.90%
	Run7	62891	64079	74920	1.89%	19.13%	14.47%
	Run8	62732	64521	75061	2.85%	19.65%	14.04%
	Run9	62863	64883	74773	3.21%	18.95%	13.23%
	Run10	62704	64684	75053	3.16%	19.69%	13.82%
	Average	**62871.2**	**64330.2**	**75029.8**	**2.32%**	**19.34%**	**14.26%**
	StdDeviation	**210.5710754**	**299.7542697**	**267.9331592**	**0.67%**	**0.62%**	**0.58%**
GCC8 gcc version 8.3.0 (Debian 8.3.0-6)	*Run1*	62832	64606	72160	2.82%	14.85%	10.47%
	Run2	62957	64634	72706	2.66%	15.49%	11.10%
	Run3	62985	64680	72101	2.69%	14.47%	10.29%
	Run4	63435	64168	72011	1.16%	13.52%	10.89%
	Run5	65135	64383	71809	-1.15%	10.25%	10.34%
	Run6	62763	64668	71860	3.04%	14.49%	10.01%
	Run7	62894	64275	71986	2.20%	14.46%	10.71%
	Run8	62771	64561	72026	2.85%	14.74%	10.36%
	Run9	62706	64093	71807	2.21%	14.51%	10.74%
	Run10	63079	64368	71905	2.04%	13.99%	10.48%
	Average	**63155.7**	**64443.6**	**72037.1**	**2.05%**	**14.08%**	**10.54%**
	StdDeviation	**726.3184257**	**215.8544777**	**263.2744449**	**1.25%**	**1.44%**	**0.32%**

As can be observed, the performance tests were conducted with randpoline in the form of a GCC plugin, on the following system:

- Testplatform: Intel(R) Xeon(R) Platinum 8180M (https://ark.intel.com/products/120498/Intel-Xeon-Platinum-8180M-Processor-38-5M-Cache-2-50-GHz-)
  Operating System: Debian GNU/Linux 9.6 (stretch), Linux 4.9.0-7-amd64 x86-64

The plugin was compiled with gcc versions 6.5.0, 7.3.0 and 8.3.0. The database application SQLite3 version 3.29.0 2019-06-17 14:50:33 (sql3-version command output: 45bfcb88e71451a656982e217375e257fc8e68374349c2984be1266bf86falt1) was used as the test target. We share the results without the insert int3 option of the plugin, but the performance difference is negligible in our tests.

Tests were conducted, using the TCL test harness that comes with the SQlite3 source repository. More information on SQlite3 TCL tests can be found in the testing documentation https://www.sqlite.org/testing.html. The test log file states that one test run performs a total of 248322 individual tests that belong to, as the documentation states, a total of 40870 distinct test cases.

In this particular implementation, we used both a compile-time-generated random offset and a run-time-generated random offset, in order to create a per-execution randomized randpoline offset for each indirect branch. The runtime random value used is stored in register r13 to avoid memory access latency. The actual offset into the randpoline is the sum of these two random offsets with each covering a maximum of 50% of the randpoline area, so the sum of the two does not exceed the randpoline area. In our setup, each randpoline offset is hence made up of a compile time random number, and a runtime random number, where the runtime random

number is generated once at application initialization time. If desired, the runtime random number can be regenerated throughout runtime.

This version has some variation from the simplest randpoline implementation following proposals by our reviewers, and is thus more preferable to be used for collecting performance data.

The randpoline plugin is currently implemented as an RTL pass for GCC, running after GCC's own "vartrack" pass. Additionally, the test setup requires that we remove a register from libc and other dependencies to hold our runtime random value, for which we use another very simple plugin. To run tests for randpoline, it is required we recompile libc, zlib and tcllib, to remove all use of r13, which we dedicated to be our random value register. It is important to note, the unpatched and the retpoline patched versions of sqlite3 were executed using the same setup, except without the use of any of our GCC plugins, hence with the r13 register available.

The number of patched indirect jmp/call instructions varies between compiler versions, between 539 and 553, using O2 as optimization setting. In the measurement experiments, only the program binaries were instrumented with the mentioned mitigations. For the retpoline cases, the system used does not fully enable retpoline (the OS does not do RSB stuffing, etc). The level of compiler optimization for all test results in the above table was set to O2. The numbers of indirections patched per optimization level from O0 to O3, as well as unpatchable indirect branches, are also measured and listed below:

Optimization level	Fixes	Misses
0	261	0
1	277	1
2	553	2
3	857	5

Randomization at compile time as well as runtime is achieved by using a seed-based random number generator, more specifically the rand() implementation of stdlib on Linux, with the srand() value (to initialize the seed) obtained by a call to *rdrand*.

7. Q & A

1. Does randpoline generate a misprediction for every *call/jmp*?

Not necessarily. Randpoline does not force misprediction, but instead tries to avoid the branch predictor from generating a prediction in dangerous cases.

2. Does randpoline expose a new attack surface for control flow hijacking by adding a continuous memory area with, for example, '*jmp rbx*'?

In order to leverage jumps in a randpoline pad, an attacker would first still need to exploit a memory corruption vulnerability to (1) control the rbx value and (2) redirect the control flow to the randpoline area. We believe the primary difference it may cause is in the corner case of a "partial write" attack primitive, the attacker may still be able to steer the control flow into randpoline area by partially overwrite a function pointer.

3. Does randpoline require the use of the RDRAND instruction?

No, randpoline can use other sources of entropy, for example, a fast pseudo-random number generator or even an array of pre-initialized random values. The security properties, however, will be different depending on the source of entropy that is used.

4. Does randpoline cause instruction cache misses and thus big performance overhead?

It is possible that randpoline will introduce some instructions cache misses yet our testing has not shown any significant overhead in performance.

5. Does randpoline require run-time code generation?

No, randpoline only requires additional code to be generated during compile time.

6. Which predictors randpoline applies to?

Randpoline is not predictor specific. We believe that the Randpoline idea should work in any predictor, with different levels of entropy (based on the tag bits used).

7. How randpoline affects ROP detection technologies (based on heuristics)?

Since Randpoline does not touch '*ret'*, we believe that it should not impact such technologies. Randpoline does double the number of indirect branches executed in a given code base, but even JOP detection technologies should not be affected.

8. Is it possible that the indirect jumps in the randpoline area will be 'slow' and thus speculate (based on the BTB)?

Consuming a BTB entry requires matching the branch address (or aliased address) and, in most cases, the branch history as well, which are both randomized by randpoline. Thus in theory it is still possible to hit an attacker-injected BTB entry, but with a very small probability given the randomization.

9. Is it correct to compare what randpoline provides with ASLR, in which you can 'break' by being able to control a memory area larger than the maximum ASLR shift (in which case there is no need to guess anything, no matter the randomization)?

As mentioned in section 5, BTB is an array with limited size and gets dynamically updated, so it is not possible to populate and maintain control of all entries that cover all the possible branches in the randpoline area

10. In Section 4, retpoline is said to have integration challenges with CFI, with RAP mentioned in the list. What is the compatibility problem (other than a non-trivial implementation)?

Indeed the problem is not RAP specific. Retpoline opens a race window that didn't exist before (checking anything about the return address before the actual *'retn'* is inherently racy and a way to fix it is by converting *'retn'* to an indirect jump, the very thing that retpoline wants to avoid), but it affects the security of other CFI approaches as well, not just that of RAP.

11. If *rdrand* is constantly used by the plugin, what guarantees it does complete before any indirect branch is executed?

First, the randomization can include a serializing instruction (such as *'lfence'*) to guarantee the completion of the generation. Second, if the implementation uses very frequent re-randomization (e.g. per branch), it is more desirable to use a fast pseudo random number generator with a strong seed to optimize the process. Or use a less frequent the randomization (e.g. per process, per routine or every N*times that a branch gets executed), in which case the rand offset can be properly initialized and kept in a register.

12. Is it possible that the randpoline slides through the entire area speculatively, increase the chance of hitting an attacker-injected BTB entry??

While we did not manage to reproduce such a condition, in the initial feedbacks collected from security researchers, this point was discussed and we've decided to add a *'ud2'* between the jumps in the ladder, to mitigate this condition. The plugin has an optional parameter to disable this option and we've included in this paper the performance differences between configurations.

13. The idea gives space to many different implementation possibilities, it would be good to have a simple example of another possible implementation

We consider the implementation of randpoline an open research field. Besides the per process, per routine or per branch re-randomization, it is also possible to periodically re-randomize the offset into the randpoline area based on the count of indirect branches executed. For example, a counter and re-randomization logic can be added between the original branch location and the

jump into randpoline. The counter can be implemented using a register (e.g. some bits of r13) that is incremented/decremented on every randpoline execution. Once a threshold for the counter is hit, a re-randomization code would execute and update the random offset into randpoline. The counter threshold can be adjusted based on both the security and performance concerns.

An example flow of such implementation is shown below:

(1) Direct jump from original branch location to Counter-and-Re-randomization logic
(2) Counter-and-Re-randomization logic:
    a. Increment the counter
    b. Re-randomize offset into randpoline if threshold is hit
    c. Jump into randpoline with the offset
(3) Randpoline jump to the original branch target

The implementation of re-randomization can also be time-based if desired by simply replacing the indirect branch counter with a Time Stamp Counter.

8. COMPARISON TABLE

	Retpoline	**Randpoline**	**IBRS**
Mitigates Spectre V2	Yes	Yes	Yes
Performance Cost	Improved performance compared to Baseline	Potentially additional improvements over Retpoline, due to less misprediction	Baseline
Compiler help needed	Yes	Yes	No
Microcode update	Yes (for BDW and later)	Yes (for BDW and later)	Yes, required
gcc/llvm --indirect-thunk user	Yes	Yes	No
Requires serialization in spec. path	Yes	No	No
Misprediction at use	Always	Not always, implementation specific	No
iCache miss at use	No	Depending on the randpoline pad size	No
BTB Training	No effect	No deterministic effect	No effect
Requires "source code ecosystem"	Yes	Yes	No
RDRAND	Not needed	Desired	Not needed
BTB sanitized	No	No	Yes, during IBRS transition
Indir-Branch via RET	Yes	No	No
Compatible with CET	No	Yes	Yes

Acknowledgements

We really appreciate the feedbacks and collaboration with the researchers Brad Spengler (grsecurity) and pipacs (PaX project). We also highly appreciate the feedbacks from Matt Miller, as well from two other researchers that preferred to remain anonymous. We also appreciate the feedback from researchers within STORM and the many researchers that open-sourced their GCC plugins (especially Emese Revfy and the developers of Kguard at Columbia University).

Legal Notices & Disclaimers

Randpoline is disclosed as a security research work for the community to consider, leverage and review. While its effectiveness has been demonstrated in the tests conducted, it has not been comprehensively tested against all potential corner cases and therefore is not presented nor guaranteed as a mature solution for branch target injection mitigation.

Intel provides these materials as-is, with no express or implied warranties. No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All products, dates, and figures specified are preliminary, based on current expectations, and are subject to change without notice.

Intel, processors, chipsets, and desktop boards may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration**. No product or component can be absolutely secure**. Check with your system manufacturer or retailer or learn more at http://intel.com.

Some results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

Intel and the Intel logo are trademarks of Intel Corporation in the United States and other countries.

*Other names and brands may be claimed as the property of others.

REFERENCES

1. Jann Horn. “Reading privileged memory with a side-channel”. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
2. Ben Gras; Kaveh Razavi; Erik Bosman; Herbert Bos; Cristiano Giuffrida. “ASLR on the Line: Practical Cache Attacks on the MMU”. https://www.cs.vu.nl/~herbertb/download/papers/anc_ndss17.pdf
3. PaX Team. “RAP: RIP ROP”. https://pax.grsecurity.net/docs/PaXTeam-H2HC15-RAP-RIP-ROP.pdf