2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586089

New Predictor-Based Attacks in Processors

Shuwen Deng and Jakub Szefer
Department of Electrical Engineering
Yale University
New Haven, USA
{shuwen.deng, jakub.szefer}@yale.edu

***Abstract*—The microarchitectural state held by predictors in modern processors can leak sensitive information. This is the first work to analyze the security of a special type of predictor, the value predictor, and demonstrate new security attacks. The new attacks bypass all the existing predictor defenses which have not yet considered value predictors as sources of vulnerabilities. This work further shows there are many value predictor attack variants, as derived using our new attack model. This paper highlights the importance of security analysis of processor features before they are realized in silicon, so the security is understood at the design time.**

I. Introduction

Over the last few years, security attacks on processors have demonstrated the dangers of introducing performance-enhancing features that end up contributing to information leaks. Many performance-enhancing features found in current processors have not been thoroughly analyzed for security vulnerabilities, leading to various attacks, e.g., [4], [14]. Especially, a variety of predictors such as branch predictors or prefetchers have been found to be sources of security vulnerabilities, e.g., [4]. The microarchitectural state held by these predictors can be abused through various timing-based attacks to leak secrets, such as cryptographic keys, e.g., [4]. Further, speculative execution attacks such as Spectre [7] also leverage predictors as part of the speculative attacks to recover the secrets. Understanding different types of predictors, in the context of Spectre or by themselves, is important to ensure the security of modern processors.

This is the first work to focus on understanding a special type of predictor, the value predictor, and demonstrating new security attacks on these predictors. Although not implemented in the real hardware today, numerous value predictor architectures have been proposed and are being considered for inclusion in future processors. Since the original last value predictor [8], a number of improvements have been developed, e.g., [11], including recent work in the last two years, e.g., [12]. These value predictors have demonstrated to improve processor performance, however, as we show, they can contribute to new security vulnerabilities.

By exploring value predictor attacks (and defenses), this work fills in the missing understanding of the security of value predictors. Attacks and defenses should be analyzed at design time before new features, such as the value predictors, are added to real machines. It has been shown many times before [4], [14] that attacks are found due to features introduced without proper design-time security analysis.

A. Processor Pipeline with a Value Predictor

A processor pipeline with a value predictor is shown in Figure 1. A typical value predictor [8] uses the instruction address (program counter) to keep track of the loaded values. For each instruction, once a value is predicted correctly for more than a $confidence$ number of times, the predictor starts to use the previous (last) value when a cache miss occurs. This allows instructions to proceed while the actual data value is still being fetched, providing data with high confidence and preventing the performance penalty of a cache miss. Different variants of value predictors have been demonstrated, which can improve the processors' performance from 4.8% [11] to 11.2% [9].

For a typical predictor shown in Figure 1, for each load instruction, the Value Prediction System (VPS) keeps track of the index, the data value to be predicted, and the past value history (VHist). The index can be the program counter (PC), or the data address, depending on the type of value predictor. The VPS typically further uses the full address as the index, e.g., [12]. Using a subset of the address bits is possible, but will introduce conflicts between different addresses and reduce the prediction rate. On each load, the value history and predicted value are updated. If the predicted value is verified to be correct (after the actual load data is available), the $confidence$ and $usefulness$ values are increased and there are no changes to the original load and dependent instructions. Meanwhile, a misprediction will cause not only the predicted load but also dependent instructions to be squashed and reissued. Within the VPS, if there are not enough entries, the entry with the smallest $usefulness$ value will be evicted.

B. Security Threats due to Value Predictors

To understand attacks on value predictors, one of the contributions of this work is the approach for analyzing the behavior of value predictors. When analyzing attacks, we consider a sender and a receiver processes who can 1) train the predictor, 2) modify the predictor by re-training, 3) trigger the predictor, then 4) encode the secret obtained through prior steps into a covert channel, and finally 5) decode the secret from the channel. The first three steps deal with manipulating the state of the predictor to learn the information. Meanwhile, encode and decode steps deal with *volatile*, *persistent*, or *timing-window* microarchitectural channels to reveal the secret. In particular, we believe this to be the first work to consider timing-window microarchitectural channels that leverage timing differences of memory accesses when no value prediction vs. a correct value prediction is made. Unlike other types of predictors, such as prefetchers, value predictors do not only have correct and incorrect predictions but also have no prediction (if the $confidence$ level is not reached) timing variants. We show in our attacks for the first time that no prediction vs. correct prediction can be abused to leak secrets.

C. Contributions and Paper Outline

The contributions of this work are as follows:

1) Present threat model and approach for evaluating the value predictor security (Section II).

* This work was supported in part by NSF grant 1813797 and SRC award number 2844.001. Shuwen Deng was supported through the Google Ph.D. Fellowship.

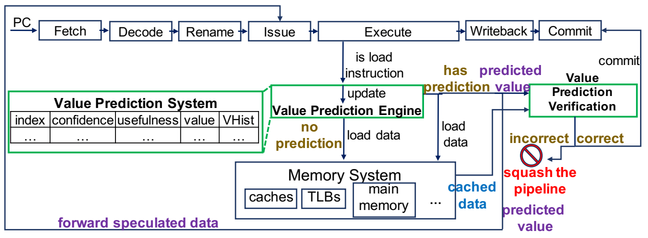


Fig. 1: Processor pipeline with a Value Prediction System (VPS).

2) Demonstrate the first, new attacks on value predictors, which can be used to attack applications and bypass existing protection schemes (Section IV).
3) Develop the first, systematic model for analyzing value predictor attacks, used to demonstrate different variants of attacks that can leverage value predictors to leak information (Section V).
4) Propose and evaluate three security techniques for securing value predictors (Section VI).

The code developed for this work will be made available under an open-source license at https://caslab.csl.yale.edu/code/value-predictor-security/.

II. Threat Model

We present the first threat model for analyzing value predictors. The model assumes there is a sender (victim) process that has access to a secret and a receiver (attacker) process which aims to learn the secret. Two processes can execute on the same core or different cores. Especially, in internal-interference attacks (which involve only the sender's accesses), two processes do not need to share the value predictor, as long as the receiver can observe timing differences in the execution of the sender (as affected by the value predictor's state or use of the value predictor). The attacker is assumed to know the source code of the victim process which is not secret by itself. The attacker further can trigger the value prediction by meeting the right condition, e.g., making $confidence$ number of accesses, or other condition used by the VPS.

We assume predictors can be broadly *PC-based predictor* (use the program counter, i.e., instruction address, for indexing the predictor's state) or *data-address-based predictor* (use the address of the accessed data to index the predictor's state) and we assume the address is a virtual address[1]. The address can also incorporate other information, such as a process identifier, *pid*, if the value predictor uses that for indexing the predictor's state. We use the term *index* to describe the information used to index the predictor's state. I.e., in PC-based predictors, the index is the PC plus any potential identifiers.

This work focuses on load-based VPS[2], where 1) training[3] or 2) modifying the value predictor state, or 3) triggering the value predictor to make a prediction, requires a cache miss. The miss is assumed to occur naturally as part of the code's execution or can be forced by a malicious attacker that invalidates or flushes the cache. Having value prediction at the frontend or in the execution stage of the pipeline will not influence the attacks we propose in this work, since the attack mechanism is independent of stages in the pipeline where the VPS is used, as long as the prediction can happen before the actual value is obtained.

[1]Physical-address-based attacks are also possible, but we focus on attacks using virtual address as most value predictors we studied use virtual addresses.

[2]Non load-based VPS is possible, where the attacks can be triggered without causing cache misses, discussion of such VPS is omitted due to limited space.

[3]To train the predictor to make a prediction, we assume a *confidence* number of accesses are required. Thus, the predictor will output a first prediction on the $confidence + 1$ access.

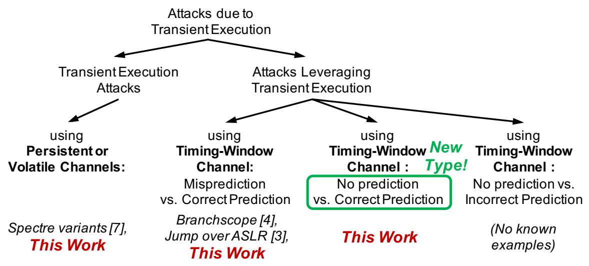
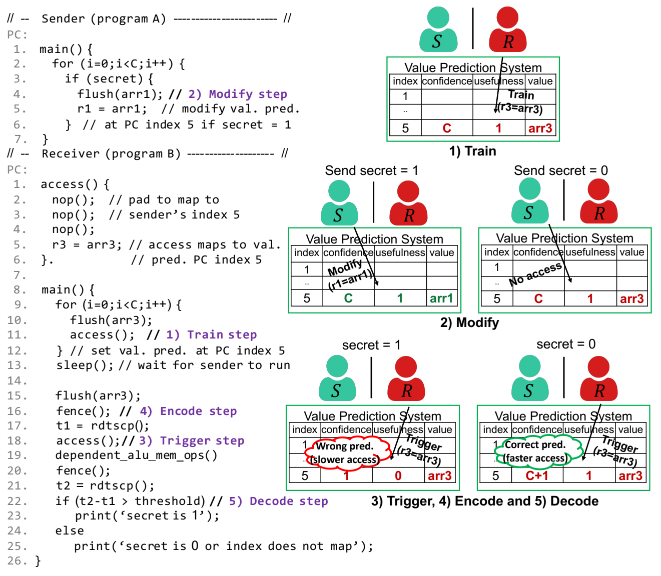

Attacks due to Transient Execution
Transient Execution Attacks
Attacks Leveraging Transient Execution
using Persistent or Volatile Channels:
using Timing-Window Channel: Misprediction vs. Correct Prediction
using Timing-Window Channel : No prediction vs. Correct Prediction
New Type!
using Timing-Window Channel : No prediction vs. Incorrect Prediction
Spectre variants [7], This Work
Branchscope [4], Jump over ASLR [3], This Work
This Work
(No known examples)

Fig. 2: Taxonomy of timing-window microarchitectural channels. We are the first to present a no prediction vs. correct prediction timing attack.



```
// --  Sender (program A) ---------------------- //
PC:
 1.  main() {
 2.    for (i=0;i<C;i++) {
 3.      if (secret) {
 4.        flush(arr1); // 2) Modify step
 5.        r1 = arr1;  // modify val. pred.
 6.      }  // at PC index 5 if secret = 1
 7.  }
// --  Receiver (program B) ------------------ //
PC:
 1.  access() {
 2.    nop();  // pad to map to
 3.    nop();  // sender's index 5
 4.    nop();
 5.    r3 = arr3; // access maps to val.
 6.  }.             // pred. PC index 5
 7.
 8.  main() {
 9.    for (i=0;i<C;i++) {
10.      flush(arr3);
11.      access();  // 1) Train step
12.    } // set val. pred. at PC index 5
13.    sleep(); // wait for sender to run
14.
15.    flush(arr3);
16.    fence(); // 4) Encode step
17.    t1 = rdtscp();
18.    access();// 3) Trigger step
19.    dependent_alu_mem_ops()
20.    fence();
21.    t2 = rdtscp();
22.    if (t2-t1 > threshold) // 5) Decode step
23.      print('secret is 1');
24.    else
25.      print('secret is 0 or index does not map');
26. }
```

Fig. 3: Proof-of-concept code and diagram of the value predictor's state for a Train + Test attack.

III. Attacks Taxonomy

Figure 2 shows the taxonomy of transient execution attacks and attacks leveraging transient execution. Attacks leveraging transient execution modify transient execution behavior based on the secret value. For this type, there are timing-window attacks that rely on misprediction vs. correct prediction timing, e.g., Branchscope [4], Jump over ASLR [3], or one of our attack variants. We also show a different attack variant that uses no prediction vs. correct prediction timing-window attack, which is a different, new type of attack. No prediction vs. incorrect prediction attacks theoretically exist but such types are not known. In addition, we also show value predictor attack variants that can be used with regular transient execution attacks.

IV. New Value Predictor Attacks

In this section, we demonstrate two new proof-of-concept attacks on value predictors.

A. Train + Test Attack

The first new attack we present is the Train + Test attack, as is shown in Figure 3. In this attack, the attacker (receiver) is able

```
// --  Sender (program A) --------------------------- //
PC:
 1.  main() {
 2.    for (i=0;i<C;i++) {
 3.      flush(secret_bit);
 4.      r1 = secret_bit; // 1) Train step
 5.       // set pred. at PC index 4
 6.  }
// --  Receiver (program B) ---------------------- //
PC:
 1.  check_hit() {
 2.    nop();          // pad with nops
 3.    flush(known_bit);
 4.    x = known_bit; // access maps to val.
 5.    return x;       // pred. PC index 4
 6.  }
 7.
 8.  main() {
 9.    sleep(); // wait for sender to run
10.
11.    flush(arr2);
12.    x = check_hit(); // 3) Trigger step,
13.    y = arr2[x*512]; // 4) Encode step,
14.    // x will be accessed speculatively
15.    // and modify cache based on x
16.    sleep(); // wait for spec. exe.
17.
18.    for (j=0;j<256;j++)
19.      t1 = rdtscp();  // 5) Decode step,
20.      r2 = arr2[j*512]; // check which
21.      t2 = rdtscp(); // entry was modified
22.
23.     if (t2-t1 < threshold)
24.       print('secret_bit candidate is: %x', j);
25.    }  // print secret read from cache channel
26. }
```

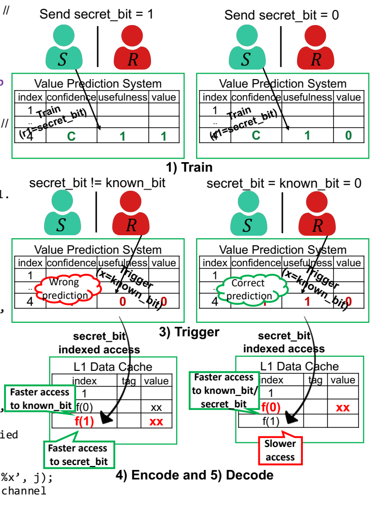


Fig. 4: **Proof-of-concept code and diagram of the value predictor's state for a Test + Hit attack.**

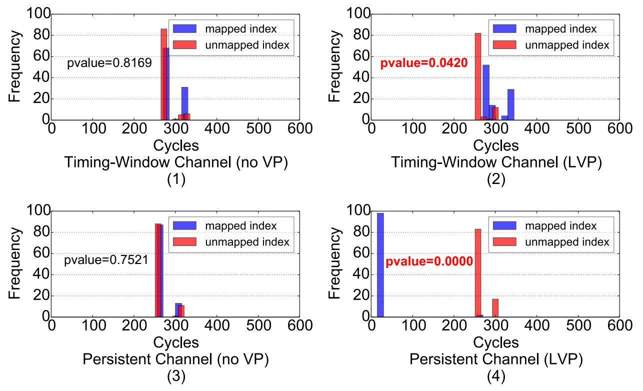


Fig. 5: Timing distribution results of Train + Test attacks using timing-window channel (1-2) and persistent channel (3-4). Red *pvalue* means the related attack is effective, while black means it is not.

to derive the value predictor index accessed by the victim (sender) during a load operation, and with the knowledge of the source code, the receiver can correlate the index accessed to the secret value 1 or 0 they are trying to learn. In this attack, first, the predictor is set to a known state in the train step by having a $confidence$ number of accesses to a known index. In the modify step a further $confidence$ number of secret-related access can be made to set a new valid predictor state or 1 access can be made to cause the previously trained value to be invalidated. In the final trigger step, there is 1 memory access to a known index, as in the first step.

If modify step involves $confidence$ number of accesses, as is shown in Figure 3, there will be a correct prediction in the last step if secret and known indices are different or *secret* is 0, since the predictor state was not modified by the middle modify step.[4] There will be a misprediction if indices are the same and the *secret* is 1 (predictor state was modified by the middle step that maps to the same index as the known index steps). If the modify step has 1 access and secret-dependent access by the sender maps to the same index as the known index access, it resets the $confidence$ value to 0 and leads to no prediction in the last step. Otherwise, there will be a correct prediction as no modification of states in the middle step.

B. Test + Hit Attack

The second attack we present is a Test + Hit attack, shown in Figure 4. In this attack, the receiver is able to derive the *secret_bit* value that was trained into the value predictor state by the sender's accesses. First, the sender accesses the secret at least a $confidence$ number of times to train the predictor. The receiver can for example force the sender to repeatedly execute the code that uses the secret value and causes it to be trained into the value predictor state. Next, the receiver makes access in the trigger step (no modify step is used in this attack) to a known data at the same index as the sender did. The access triggers the value predictor to make a prediction related to the secret value. When prediction occurs, during transient execution the output of the value predictor can be encoded into the persistent cache channel. Similar to Spectre attacks, array access is performed in Figure 4, where the index is the value from the value predictor. To recover the secret value from the cache channel, the receiver checks the timing of accessing the array elements, to learn which one was previously placed into the cache and thus recover the secret.

[4]There can be a correct prediction also if the indices are the same and the secret data and known data happen to be the same. However, for index-focused attacks there is no assumption about data knowledge and the probability of this is approximately 1^{-64} for 64-bit data. Further, if this attack involves known data, it becomes equivalent to the Test + Hit attack.

C. Experimental Setup

To evaluate new value predictor attacks, we implemented value predictors in a modified `gem5` simulator [2], and run the code on the simulator. The `gem5` simulator was used in *syscall emulation* mode (SE) with the O3CPU model and Ruby cache for testing. We implemented a baseline (non-secure) LVP [8] predictor. and an oracle VTAGE [10]. The oracle value predictor makes predictions only for the target load instruction to maximize the attacker's advantage. To judge whether an attack is successful, we report averages over 100 runs for each attack, with a 95%-confidence interval [1] calculated using the Student's t-test [5] to distinguish measured timing distributions.

D. Attack Evaluation and Results

For our evaluation, we focus on analyzing if the receiver can distinguish two types of timings – "mapped" vs. "unmapped" cases – as explained below. We use *pvalue* calculation result to determine if two types of timings can be distinguished. If the *pvalue* is smaller than 0.05, timing distributions are differentiable and the attack succeeds.

1) Train + Test Attack Results: For the timing-window channel, when the secret and known indices map to each other and *secret* is 1, misprediction leads to longer timing in the trigger step. Meanwhile, there will be a correct prediction in the trigger step when not mapped, since the predictor state set in the train step was not modified.

We also evaluate a persistent channel variant of this attack, where mapped case means two indices are the same and *secret* is 1, resulting in misprediction in the trigger step, which encodes the data into the cache, and a cache hit is observed in the reload part of the covert channel. Otherwise, the unmapped case results in a cache miss.

In Figure 5 (1) and (3) it can be seen that without value predictor (no VP), different timing distributions cannot be distinguished, and the attacks are not possible. Meanwhile, in Figure 5 (2) and (4) it can be seen that when (non-secure) LVP value predictor is enabled, timing distributions for mapped and unmapped cases are different, and the secret value can be leaked.

```
1. void _gcry_mpi_powm (gcry_mpi_y_ res,
2.          gcry_mpi_t base, gcry_mpi_t expom gcry_mpi_t_mod)
3. {
4.     mpi_ptr_t rp, xp; /* pointers to MPI data */
5.     mpi_ptr_t tp;
6.     ...
7.     for(;;) {
8.          /* For every exponent bit in expo*/
9.          _gcry_mpih_sqr_n_basecase(xp, rp);
10.         if(secret_exponent || e_bit_is1) {
11.            /* unconditional multiply if exponent is
12.             * secret to mitigate FLUSH+RELOAD
13.             */
14.            _gcry_mpih_mul(xp, rp);
15.         }
16.         if(e_bit_is1) {
17.            /*e bit is 1, use the result*/
18.            tp = rp; rp = xp; xp = tp;
19.            rsize = xsize;
20.         }
21.     }
22. }
```

Fig. 6: Code of modular exponentiation from `libgcrypt`, adapted from [6]. The highlighted red-colored part shows conditional access to the `tp` index which can be leaked through value predictor attacks.

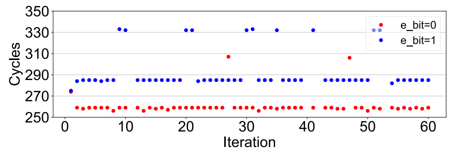


Fig. 7: Sequences of the receiver's observation for each iteration when the `e_bit` is 0 or 1 (line 16 shown in Figure 6).

Our proof-of-concept code can be extended to real applications. For example, Figure 6 shows the RSA related portion of `libgcrypt` code with conditional memory access. The code is already protected against Flush + Reload cache timing attacks [14]. However, when the value predictor is trained through repeated accesses (due to repeated invocations of the code with the same cryptographic key), the index of the `tp` access can be leaked through the value predictor attack, leaking the value of `e_bit_is1` as is shown in Figure 7. The success rate of correctly transmitting `e_bit` is 95.7% for 60 runs, which is enough to reconstruct the full key based on prior work [6]. Without further optimization, the transmission rate is 9.65Kbps.

Our attack is demonstrated on RSA modular exponentiation which does not use blinding techniques. We do this because most related work in architecture does not consider blinding and we target similar code for easier comparison. There are blinding techniques for both RSA and ECC, due to limited space, we do not discuss blinding here. However, we expect that a variant of our value prediction attack actually works if the blinding scheme is used. If the secret is accessed by a load or similar instruction during the blinding operation, we can use value prediction to extract the secret (it is not possible to extract the blinding factor, as it is random each time, while the secret is constant and gets trained into the value predictor). This type of attack is not possible with branch predictor or cache side channels but is possible to value predictor attacks.

2) Test + Hit Attack Results: For the timing-window channel, mapped data means that the data accessed in the train step and the trigger step are the same and a correct prediction can be derived in the trigger step (faster timing), while in the unmapped case the two data are different (slower timing). For the persistent channel, the mapped case means the secret value encoded by the train step is brought to the cache, causing reloading a fast timing. Therefore, the secret value can be observed through cache hits in the cache channel. Otherwise, only cache misses can be observed for the unmapped case.

As is shown in Figure 8, when no value predictor is used,

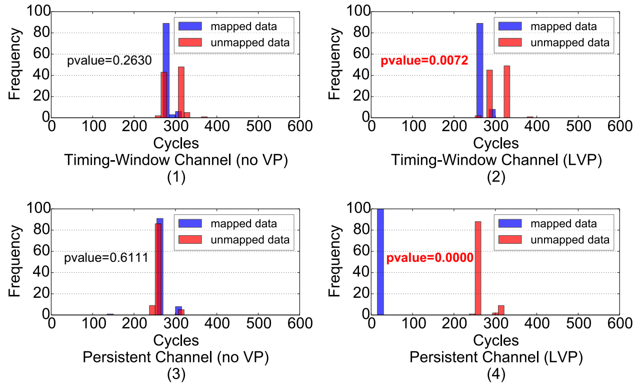


Fig. 8: Timing distribution results of Test + Hit attacks using timing-window channel (1-2) and persistent channel (3-4). Red *pvalue* means the corresponding attack is effective, while black means it is not.

there is no attack due to no significant difference in the timing distributions. Meanwhile, with (non-secure) LVP enabled, the mapped and unmapped timing distributions are significantly different.

3) Value Predictor Type Influence: We further evaluate whether the type of value predictor, e.g., LVP vs. VTAGE, has impacts on the attacks. For both predictor types, timing distributions between mapped and unmapped cases are significantly different to leak data. The VTAGE data and details are omitted due to limited space.

V. Value Predictor Attack Modeling

Based on the two proof-of-concept attacks, we further present the first model for analyzing value predictor security. The model further points to additional attack types. The model is based on exploring all possible steps that the victim or the attacker can perform to affect or observe the value predictor state, and how that can leak information.

1) Train Step: In this step, the value predictor is trained using load access at a certain index, to set up a deterministic predictor state for the PC's or data address's prediction entry. This step can be secret-related if performed by the sender who is the only one with logical access to the secret. In this case, this step is used to put the secret-related data into the predictor state so it can be revealed by other steps. Otherwise, this step is used to provide a known reference state that can be later used to derive secret information by observing state changes, which can be performed by the sender or the receiver.

To train the value predictor, the loads usually need to be accessed at least $confidence$ number of times to set the predicted state. However, for certain attacks, the access is made $confidence - 1$ number of times, so that the access in the next modify step can be detected if it pushes the total accesses to the $confidence$ number of times, and the prediction is triggered to output a value during a cache miss.

2) Modify Step: In some attacks, the modify step is needed to alter the value predictor's state set in the first step before an observation is made. This step is useful if the first step was to known data or index, and a state modification (due to secret-related access) is needed to observe potential interferences. This step is also useful if the first step is secret-related, and the state modification is due to another (possibly the same) secret-related access, or due to a known data or index access. For this step, most attacks will repeatedly execute load more than $confidence$ number of times to encode the value into the predictor's state. However, for some attacks, only 1 extra access in this step is needed if the train step uses $confidence - 1$ accesses.

TABLE I: Possible actions for each step of value predictor attacks.

Action	Description
S^{KD}, S^{KI}	Sender makes access to data, or respectively index, that it knows.
R^{KD}, R^{KI}	Receiver makes access to data, or respectively index, that it knows.
$S^{SD'}$, $S^{SD''}$	Sender makes access to secret data the receiver tries to learn. For attacks leveraging interference between sender's accesses, secret data D' and D'' may or may not be the same, which is what the receiver is trying to learn. For attacks that involve known data, the goal is to learn if the known D is or is not the same as the secret.
$S^{SI'}$, $S^{SI''}$	Sender makes access to secret-dependent index the receiver tries to learn. For attacks that involve the known index, the goal is to learn if the known I is or is not the same as the secret index.
—	This step is not used, this is only for the modify step for attacks that do not have any actions in the modify step.

3) Trigger Step: A single access is required in this step to probe the value predictor to observe the interference that can reveal the secret, or even directly observe the secret through timing variations.

4) Encode and 5) Decode Step: The sensitive information obtained from predictor states needs to be encoded into a channel to exfiltrate the information. This can be a persistent channel (e.g., cache channel), a volatile channel (e.g., port contention channel), or a timing-window channel (e.g., directly measure the timing of the load access and subsequent instructions). Depending on the types of three possible channels used, related ways are used to decode the secret.

A. Modeling Results

To understand possible attacks, we consider the first three steps, as the last two steps are about exfiltrating the information, and are not specific to value predictors. The first three steps can be performed by the sender S or the receiver R. The possible actions in each step are shown in Table I. With these actions, there are in total 576 possible three-step combinations for the train, modify, and trigger steps: 8 step types for train step (S^{KD}, S^{KI}, R^{KD}, R^{KI}, $S^{SD'}$, $S^{SD''}$, $S^{SI'}$, and $S^{SI''}$) $\times$ 9 step types for modify step (the same as train step plus —) $\times$ 8 step types for trigger step (the same as train step) = 576 combinations. However, the majority of these 576 combinations do not represent attacks or can be reduced to simpler patterns. We define rules to determine if a pattern corresponds to a possible attack, and eventually show that there are exactly 12 effective attacks, as discussed in Section V-B. Rule description and soundness analysis of the model are not included due to limited space.

B. Value Predictor Attack Variants

Following our analysis, there are 12 value predictor attack variants. The attacks are summarized below and shown in Table II. If the predictor indexing function uses pid or another identifier, and two known data or index steps are done by different processes (not both S nor both R), then the known data or index has to come from the shared library so both can access the same index. However, if the index is just based on the address and no pid (as is the case of many known value predictors [9]–[12]), then no shared library is needed.[5]

1) Train + Test: Details of this attack were given in Section IV.

2) Test + Hit: Details of this attack were given in Section IV.

3) Train + Hit: This is a two-step attack where in the train step the predictor state is first set by the $confidence$ number of accesses to a known data. Next, 1 secret-related access is made. Correct prediction makes execution faster which shows that the known data is the same as secret-related data, otherwise execution is slower and two data

[5]Using pid only increases difficulties for attacks but does not eliminate it.

TABLE II: List of value predictor attacks and attack categories that each attack belongs to. Each step is explained in Section V-A. Each attack category is explained in Section V-B.

Step 1 (Train)	Step 2 (Modify)	Step 3 (Trigger)	Attack Cateogry
S^{KD}	—	$S^{SD'}$	Train + Hit
S^{KI}	$S^{SI'}$	S^{KI}	Train + Test
S^{KI}	$S^{SI'}$	R^{KI}	Train + Test
R^{KD}	—	$S^{SD'}$	Train + Hit
R^{KI}	$S^{SI'}$	S^{KI}	Train + Test
R^{KI}	$S^{SI'}$	R^{KI}	Train + Test
$S^{SD'}$	$S^{SD''}$	$S^{SD'}$	Spill Over
$S^{SD'}$	—	S^{KD}	Test + Hit
$S^{SD'}$	—	R^{KD}	Test + Hit
$S^{SD'}$	—	$S^{SD''}$	Fill Up
$S^{SI'}$	S^{KI}	$S^{SI'}$	Modify + Test
$S^{SI'}$	R^{KI}	$S^{SI'}$	Modify + Test

TABLE III: Value predictor attack evaluation for all the attack categories. Red *pvalue* means the corresponding attack is effective with transmission rate shown, while black means it is not. *Tran. Rate* is the transmission rate, or bandwidth, of the attack.

Attack Category	Timing-Window Channel		Persistent Channel	
	No VP	VP (Tran. Rate)	No VP	VP (Tran. Rate)
Train + Hit	0.1620	0.0086 (7.72Kbps)	—	—
Train + Test	0.8169	0.0420 (7.38Kbps)	0.7521	0.0000 (6.88Kbps)
Spill Over	0.2989	0.0000 (8.12Kbps)	—	—
Test + Hit	0.2630	0.0072 (7.81Kbps)	0.6111	0.0000 (7.43Kbps)
Fill Up	0.3734	0.0083 (8.22Kbps)	0.4677	0.0000 (6.85Kbps)
Modify + Test	0.2966	0.0000 (8.00Kbps)	—	—

are different. A timing-window channel can be used to observe the timing difference of correct prediction vs. misprediction.

4) Spill Over: This type of attack aims to determine if two secret-related states are the same or different. First, $confidence - 1$ number of accesses are made to secret data. Next, 1 access is made to possibly the same or different secret data. Finally, in the trigger step 1 access is made to the same secret data as in the first step. The last step will be predicted correctly if all the secrets accessed are the same. Otherwise, the $confidence$ value is not reached (since the middle step accesses a different value) and there is no prediction. A timing-window channel can show the timing difference of correct prediction vs. no prediction, to learn if the values are the same or not.[6]

5) Fill Up: This two-step attack has a $confidence$ number of accesses to secret data in the train step and 1 access to secret data in the trigger step, which is possibly the same or different secret. The last step has correct prediction if two secrets match, otherwise a misprediction is derived. A timing-window channel shows timing differences of correct and incorrect prediction to learn the secret. The secret can also be extracted from transient execution using a persistent or volatile channel since the predictor is trained on the secret.

6) Modify + Test: This is a flipped image of the Train + Test attack. First, a $confidence$ number of secret-related accesses is performed in the train step. In the modify step there are $confidence$ accesses or 1 access to a known index to change or invalidate the predictor state, respectively. In the trigger step, there is 1 secret-related

[6]If all the data are the same, the secret value can be itself extracted from transient execution in the last step, but this reduces to the Fill Up attack since $confidence - 1$ plus 1 access add up to $confidence$ accesses captured by the train step in the Fill Up attack. Further, this is a weaker version of Fill Up attack, since it only leaks data if all the accesses are the same secret, while Fill Up leaks data in all the cases using a persistent or volatile channel.

index access. A timing-window channel shows the timing difference of correct vs. incorrect prediction or correct vs. no prediction.

All the attack variants discussed above can use a timing-window channel to observe the timing difference due to prediction states. Further, 2) Train + Test, 4) Test + Hit, and 5) Fill Up can use a persistent or volatile channel to extract secret data from transient execution since the predictor is trained on the secret before the trigger step. Evaluation results of all the attack categories are shown in Table III, which proves the effectiveness of all the attack variants.

VI. Secure Value Predictors

Security defenses such as InvisiSpec [13] can prevent existing transient execution attacks, but have not considered value prediction in particular, and are not effective against our new attacks. Consequently, we present value-predictor-specific defenses which shows an estimation of possible defense value predictors can consider.

A. Defense Techniques

Always predict a value (A-type) defense makes the predictor always predict the value based on a fixed value or on a history value regardless of whether confidence level is reached or not. In this case, the attacks based on differentiating from prediction vs. no prediction timing are protected. **Delay side-effects (D-type)** defense targets delaying the microarchitectural state changes and can only be used for preventing value predictor attacks based on persistent channels. **Randomly predict a value (R-type)** defense randomly predicts a value out of a window around the actual accessed value. Assuming the window size is S, the rate of randomly predicting the correct value is 1/S.

B. Defense Strategies Evaluation

When all the A-type, D-type, and R-type defenses are combined, all attacks we have considered can be defended. Note that R-type defense has a (predictable) probability of attacker learning the correct value based on the window size. This probability can be made arbitrarily small at some cost to performance.

The *Train+Test attack* can be prevented as long as the R-type defense is applied. D-type defense is effective only against the persistent-channel variant of Train+Test attack but not others. The *Modify+Test attack* can be prevented when the R-type defense is applied as well. The *Test+Hit attack* can be prevented by combining both A-type and R-type defense. D-type defense is effective against only the persistent-channel type of Test+Hit attack. The *Train+Hit attack* can be prevented by combining both A-type and R-type defense. The *Spill Over attack* can be prevented by the A-type defense directly. The *Fill-Up attack* can be prevented by R-type defense.

Due to the limited space we only provide the analysis of the Train+Test and Test+Hit attacks below.

For the Train+Test attack, it will be prevented as long as the R-type defense is applied, and optionally D-type can be applied for preventing persistent channel variants. We evaluated the influence of the window size and found that a window size of 3 is the minimal size for this type of attack to guarantee security (p-value larger than 0.05 in our experiments), while at the same time maintaining the performance. Since Train+Test differentiates correct prediction vs. misprediction, A-type defense will not work, so this defense is not helpful in this case. For attack variants that use a persistent channel, D-type defense can be used.

For the Test+Hit attack, combining A-type and R-type defense can prevent the attack. For the R-type defense, in our experiments, on Test+Hit attack, the evaluation shows that window size of 9 is the minimal size for this type of attack to guarantee security (p-value larger than 0.05). This will cause large degradation to the performance. Therefore, a smaller window size is selected to maintain performance and partial security, e.g., size of 5. In addition, adding A-type defense is required to assist in fully preventing the attacks. D-type defense can be used for the persistent channel type of Test+Hit attack, but again it by itself will not defend non-persistent-channel attack types, so both A-type and R-type should be used.

VII. Conclusion

Improving processor performance without considering security at the architecture level can lead to serious security problems that have been publicized in recent years, especially due to information leaks and timing channels. One of the processor features proposed for performance improvement is value prediction, but prior to this work, their security has not been evaluated. As we show in this work, value predictors are indeed vulnerable to information leaks. We also presented three security features to protect value predictors from different attacks, and suggest defenses that should be used when value prediction is implemented in real processors.

References

[1] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting Speculative Execution through Port Contention," in *Conference on Computer and Communications Security*, 2019, pp. 785–800.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[3] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR," in *International Symposium on Microarchitecture*, 2016.

[4] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018, pp. 693–707.

[5] W. S. Gosset, "The Probable Error of a Mean," *Biometrika*, pp. 1–25, 1908.

[6] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks," in *USENIX Security Symposium*, 2018, pp. 955–972.

[7] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre Attacks: Exploiting Speculative Execution," in *Symposium on Security and Privacy*, 2019, pp. 1–19.

[8] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction," in *International Conference on Architectural Support for Programming Languages and Operating System*, 1996, pp. 138–147.

[9] A. Perais and A. Seznec, "BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction," in *International Symposium on High Performance Computer Architecture*, 2015, pp. 13–25.

[10] A. Perais and A. Seznec, "Practical Data Value Speculation for Future High-End Processors," in *International Symposium on High Performance Computer Architecture*, 2014, pp. 428–439.

[11] R. Sheikh, H. W. Cain, and R. Damodaran, "Load Value Prediction via Path-based Address Prediction: Avoiding Mispredictions due to Conflicting Stores," in *International Symposium on Microarchitecture*, 2017, pp. 423–435.

[12] R. Sheikh and D. Hower, "Efficient Load Value Prediction using Multiple Predictors and Filters," in *International Symposium on High Performance Computer Architecture*, 2019, pp. 454–465.

[13] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *International Symposium on Microarchitecture*, 2018, pp. 428–441.

[14] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." in *USENIX Security Symposium*, 2014, pp. 719–732.