WarpAttack: Bypassing CFI through Compiler-Introduced Double-Fetches

Jianhao Xu*†, Luca Di Bartolomeo†, Flavio Toffalini†, Bing Mao*, Mathias Payer†
State Key Laboratory for Novel Software Technology, Nanjing University, EPFL†*

***Abstract*—Code-reuse attacks are dangerous threats that attracted the attention of the security community for years. These attacks aim at corrupting important control-flow transfers for taking control of a process without injecting code. Nowadays, the combinations of multiple mitigations (e.g., ASLR, DEP, and CFI) drastically reduced this attack surface, making running code-reuse exploits more challenging.**

Unfortunately, security mitigations are combined with compiler optimizations, that do not distinguish between security-related and application code. Blindly deploying code optimizations over code-reuse mitigations may undermine their security guarantees. For instance, compilers may introduce double-fetch vulnerabilities that lead to concurrency issues such as Time-Of-Check to Time-Of-Use (TOCTTOU) attacks.

In this work, we propose a new attack vector, called WarpAttack, that exploits compiler-introduced double-fetch optimizations to mount TOCTTOU attacks and bypass code-reuse mitigations. We study the mechanism underlying this attack and present a practical proof-of-concept exploit against the last version of Firefox. Additionally, we propose a lightweight analysis to locate vulnerable double-fetch code (with 3% false positives) and conduct research over six popular applications, five operating systems, and four architectures (32 and 64 bits) to study the diffusion of this threat. Moreover, we study the implication of our attack against six CFI implementations. Finally, we investigate possible research lines for addressing this threat and propose practical solutions to be deployed in existing projects.

1. Introduction

Most software running on today's systems is written in low-level languages like C or C++. As these languages are prone to memory corruption bugs that often enable attackers to launch powerful code execution attacks, the applications need to be thoroughly tested to remove as many bugs as possible. As testing is generally incomplete due to state explosion, mitigations are added to the code to make exploitation more challenging [1].

Widely deployed mitigations such as Address Space Layout Randomization (ASLR) [2], stack canaries [3], and data execution prevention (DEP) [4] lower the exploitability of bugs. However, code-reuse attacks such as Return-Oriented Programming (ROP) [5–9] remain effective under all these mitigations. These attacks redirect control flow to execute legitimate instruction sequences in program memory for malicious purposes.

Control-Flow Integrity (CFI) [10, 11], widely recognized as a key mitigation to stop code-reuse attacks, restricts control-flow transfers to strictly follow some benign program execution. Specifically, CFI first statically computes the program's control-flow graph (CFG) and determines all legitimate targets of control-flow transfers. Then, CFI instruments the code with checks to validate each control-flow transfer at run-time. CFI is practical: CFI implementations are readily available in production compilers and have been widely deployed, with some trade-offs between granularity and performance. For example, the Android ecosystem deployed LLVM-CFI [12, 13] in Android 9's kernel [14], and Windows applications are widely protected by Microsoft's Control Flow Guard [15].

Despite the advance of modern CFIs, they cannot stop all the attacks. In practice, not all code may be protected through CFI [16] since their precision is limited by the target sets size. In an ideal scenario, only one possible target is valid for each indirect control-flow transfer, however, CFI checks indirect transfers against a set of targets. Therefore, attackers may choose any target in this set, allowing some deviation from a benign execution. This deviation may be sufficient for an attack. Still, the key advantage of CFI is that the size of the target set is generally small (between many targets with under ten targets and few targets at around 100 targets even for complex applications) [17]. Without CFI any executable byte is a valid target, resulting in millions of targets. CFI is therefore believed to practically stop code reuse attacks or, at least, make it almost impossible to bypass. Mitigations are generally implemented as compiler passes (or, rarely, as static binary rewriting passes). However, compilers currently are *unaware of security checks* inserted by mitigations and consider them just code that undergoes the same optimizations as all non-privileged code. For example, the compiler may spill sensitive variables used for security checks [18] that suffer from double-fetch vulnerabilities (a variable is read multiple times without integrity checks). When these cases appear, they may leave room for a race condition and resulting in Time-of-Check to Time-Of-Use [19] (TOCTTOU) opportunities for attackers.

In this work, we present WarpAttack: a novel attack that bypasses strong anti-code-reuse mitigations and grants arbitrary code execution to adversaries. Our attack exploits a misalignment between CFI implementations and assumptions of C/C++ compilers when translating switch statements to jump tables (or multiple checks against one target). The CFI threat model assumes that adversaries can modify arbitrary data including function pointers but not

code [20–23]. Therefore, CFI implementations assume that a combination of storing jump tables in read-only memory and bound checking the indirect control flow transfer will sufficiently protect the jump tables. However, compilers are unaware of CFI's security assumptions and optimize code, which results in double fetch vulnerabilities and TOCTTOU attacks between the bound check and the indirect jump. The code for computing indirect jumps in switch statements often follows this pattern: given a jump table and an index, the program first fetches the index value to validate the bound check against the jump table size, then, it fetches the index value again for processing the actual jump. Since the index is loaded (fetched) multiple times, an adversary may race against the bound check and overwrite the index value after the bound check itself, finally allowing arbitrary jumps outside the jump table. The attacker may lose the race by changing the target value too early or too late. If the value is changed before the bounds check, the switch statement will likely revert to the default case. If the value is changed after the jump, it will likely overwrite a stale value. Similar issues were investigated by previous reports [24], that describe double-fetches near bound-checked indirect jump to a jump table but, so far, were considered a concurrency bug [25].

Through our study, we research TOCTTOU security issues introduced by double-fetches. More precisely, we aim at demonstrating: (a) how compilers regularly insert double-fetches of sensitive variables, (b) how to automatically find these cases, (c) discuss their presence in sensitive software, and (d) show practical exploits in commercial applications.

To evaluate our new attack WarpAttack, we implement (and release as open-source) a proof-of-concept attack against Firefox to demonstrate WarpAttack is practical under a realistic adversary model. Our PoC shows how to win the race condition and take control of the victim program. Additionally, we implement a lightweight binary analysis to detect compiler-introduced double-fetches in widely used C/C++ programs on multiple platforms (with a false-positive rate of 3%). Our results show that (i) mainstream compilers including Clang, MSVC, and GCC generate such victim code patterns in sensitive situations, (ii) most CFI implementations are vulnerable to our attack, (iii) widely used C/C++ programs like Firefox and Chrome present such victim code patterns; (iv) these code patterns can be found on different platforms including Linux, Windows, and Mac OS and cross-architecture (i.e., x86/64, ARM, MIPS, RISCV). Finally, we discuss possible mitigations and propose future research directions to cope with WarpAttack.

In summary, our contributions are:

- We introduce WarpAttack, a new attack to bypass code-reuse mitigations and grant arbitrary code execution to adversaries.
- We show a working proof-of-concept example of WarpAttack against Firefox.
- We conduct a comprehensive research on compiler-introduced double-fetch code samples in popular applications across multiple OSs and architectures.
- We perform a study of different compilers and CFI implementations to demonstrate the magnitude of our attack has real-world impact.
- We design (and implement) a new lightweight binary analysis to detect victim code (which is available at https://github.com/HexHive/WarpAttack).

2. Background

This section introduces background information for understanding WarpAttack.

2.1. Code reuse attacks and mitigations

Code reuse attacks use memory corruptions to overwrite indirect forward edges (e.g., call/jumps) or backward edges (e.g., return instructions) for taking control of the program execution (thus granting arbitrary code execution to the attacker). Once a code reuse payload is injected and activated, it diverges the normal execution flow of a program into one attacker-controlled. The core idea is to stitch together small pieces of code, called *gadgets*, in so-called *chains*. In the literature, there are plenty of techniques to create and inject code reuse payloads that target either backward (ROP [6]) and forward jumps (JOP [8] or vtable [26]). Regardless the technique used, the chain is injected through the memory corruption vulnerability and set such that, when the target indirect jump is traversed, the control-flow is transferred to the first gadget of the chain, finally triggering the attack.

For mitigating code reuse attacks, researchers propose solutions to guarantee the correct execution of the program at runtime (i.e., allowing only "benign" execution flows). This is usually implemented through control-flow integrity checks (CFIs), that model all the correct program executions via a control-flow graph (CFG) and enforce correct execution through different policies. Generally, we consider CFIs for protecting forward [12] or backward jumps [27]. In this work, we mainly focus on CFI for forward jumps, however, we show WarpAttack can bypass any CFI that fulfill our requirements. Enforcing perfect CFIs require inferring all the correct executions of a program, thus ensuring that (at runtime) any indirect control-flow transfer lands on one single target. However, enforcing this property is impracticable since it requires one to infer all the possible execution flows. In practice, CFIs adopt an approximation that allow indirect jumps to land over a set of valid targets (instead of a single one). Even though this leaves room for attacks [28, 29], it has been proved the target sets are small enough to make code reuse attacks generally challenging [17].

2.2. CFI bypassing methods

Current offensive research lines for bypassing CFIs focus on different angles, that we classify in two main categories: targeting policy, or targeting implementation issues. In the first category, a CFI may implement a too loose target set, that *de-facto* allows an attacker to bend the execution flow while remaining undetected. Alternatively, if the target set is strict enough and no jumps are overwritten, adversaries

proposed novel techniques, called *data-only attacks* [30], to perpetrate malicious actions without breaking the CFI policy. In the second category, attackers exploit CFI implementation errors. For instance, by taking advantage of low-level code optimizations (e.g., spilled registers [27]), that temporary move critical values (e.g., the target set) from registers to memory, thus leaving room to an adversary to overwrite the protections; or else, using code motion [31].

This work shows a novel attack to bypass CFI implementations by targeting specific low-level code optimizations that conflict with CFI security assumptions. More precisely, we hijack the execution-flow without tampering with the CFI policy.

2.3. Double-fetch

Double-fetch bugs occur when a privilege system reads a variable (from a memory location) multiple times, but the fetched value is inconsistent due to concurrency issues [19, 25, 32, 33]. Figure 1 exemplifies a double-fetch bug in which the victim thread reads (*fetches*) the variable `A` from the memory twice (i.e., `Check([A])` and `var := [A]`). Contemporaneously, an attacker thread manages to win the race condition by exploiting the time window in between the two *fetches* (i.e., `[A] := 0xdeadbeef`) and pass the victim thread check (i.e., `Check([A])`). More precisely, the first *fetch* correctly checks the value of `A`, while the use of `var` (i.e., `Use(var)`) works on the injected value `0xdeadbeef`.

Double-fetch issues have been deeply studied in the community and many mitigations/detections have been proposed as well [19, 25, 32, 33]. In general, the threat model faced by these works only consider concurrency issues, without taking in account possible memory overwrite vulnerabilities. Moreover, current detections of double-fetch bugs look for different code patterns, e.g., shared structures between processes. Conversely, we deal with very specific patterns that appear as result of compiler optimizations (e.g., register spilling), thus requiring ad-hoc studies.

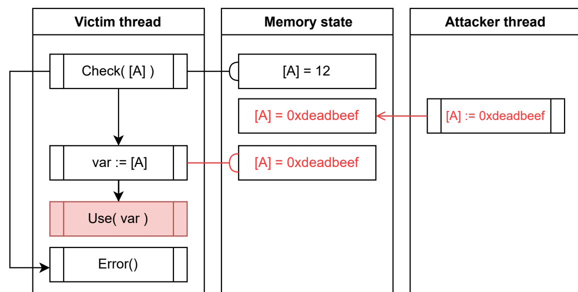

Figure 1. An example of double-fetch error.

2.4. Bound-Checked Jump Table

Compilers generally translate switch statements (and similar branching code) into jump tables. The compiler checks if the targets are dense enough, i.e., the resulting switch table would be compact with not too many empty entries. Alternatively, the compiler may use a set of branches (translating into nested if statements). A jump table allows O(1) lookup of the target while nested if statements result in O(logN) lookup time. Note that the compiler may fill empty slots and adjust the base and power of the index. For example, a switch statement with cases for 8, 16, 24 and default can be expressed by a switch table with 4 entries where the index is adjusted and shifted. The code for a jump table lookup consists of a bound check, followed by an indirect jump whose address is computed with the checked value. Most of the time, the value remains in one register, remaining immutable between the bound check and the computation of the jump address. Therefore, the indirect jump is secure in this situation, as long as the register cannot be corrupted. Mitigations against control-flow hijacking (including CFI) therefore do not pay special attention to such a bound-checked indirect jumps.

Here we give an example of such a bound-checked jump table generated from a switch statement. As shown in Figure 2, the checked object is fetched once at line 8. The fetched value in `ecx` is checked and then used at line 10 and line 13. Attackers cannot control the indirect jump as long as the register value cannot be overwritten after the check at line 10.

```
1  ;switch(obj->type) {
2  ;  case 0:
3  ; ...
4  ;  default:
5  ; ...
6  ;}
7  mov     rax, rdi
8  mov     eax, DWORD PTR [rdi+0x30]
9  add     eax, 0xffffffff
10 cmp     eax, 0x11 ;the bound check
11 ja      401163  ;default branch
12 lea     rdi, [rax+0x30]
13 jmp     QWORD PTR [rax*8+0x402008]
```

Figure 2. An example bound-checked jump table

Consecutive if statement. Besides `switch` statements, C/C++ compilers can also generate jump tables for certain code with multiple `if` statements. We refer to such situations as “Consecutive If statement”. Such kind of `if` statements may be replaced with a `switch` statement without changing the meaning of the source code. From the perspective of our attack model, we claim that the binary code from such “Consecutive if statement” is equivalent to that from a `switch` statement. That is because they are all the same at the binary level and protections of control-flow transfers like CFI do not need to specially take care of any of them.

3. Threat Model

Our threat model follows the one introduced for CFI [10] and is in line with prior offensive works [18, 26, 34]. The attacker holds the following adversarial capabilities and faces the following defensive assumptions. The goal of the attacker is to hijack control flow to a particular location.

Adversarial Capabilities.

- Arbitrary read-write: The attacker holds a memory-corruption vulnerability (such as a use-after-free error) that empowers them to read/write arbitrary memory. C/C++ programs are prone to such vulnerabilities, which is one of the reasons why CFI [10] was developed.
- Thread control: The attacker is in active control of a thread within the same address space as the program being attacked. This requirement is satisfied, e.g., when the victim browses an attacker-controlled website [35] with malicious JS code, spawning attacker-controlled threads through Web workers [36].
- Gadgets: The victim program contains sufficient gadget code snippets i.e., the switch jump table with a compiler-introduced double-fetch. This is not rare and we have found several mainstream compilers generate such gadgets with given compilation configurations.

Defensive Assumptions.

- Non-Executable Memory: The victim program enforces Data Execution Prevention (DEP) [4]. That is, the attacker cannot execute any injected "code" and must perform code-reuse attacks.
- Randomization: The victim program applies randomization protections such as address space layout randomization (ASLR [2]).
- Control Flow Protection: The victim program applies a fully precise static CFI with both forward-edge and backward-edge (such as a ShadowStack [37]) protection. CFI itself is properly configured and correctly implemented.

Difference to the CFI attacker model. Our attacker model requires two changes: (i) The adversary requires a triggerable compiler-introduced double-fetch gadget. As we show in § 7, such gadgets can be found in the x86/64 binary of many real-world C/C++ programs compiled by mainstreams compilers. Note that our attack is not restricted to compiler-based CFI but also applies to binary-only CFI, although we need compiler-introduced gadgets. That is because, ignoring orthogonal compiler optimizations, vulnerable CFI implementations do not protect bound-checked indirect jumps. (ii) Optional capabilities. Even though the CFI threat model assumes an attacker with arbitrary read capabilities. Our scenario does not strictly require this feature. Orthogonally, arbitrary read capabilities may simplify end-to-end exploits or improve the success rate.

4. Compiler-Introduced Double-Fetch

Compilers are free to optimize code according to the model of the underlying programming language semantics. For C/C++, these semantics do not discuss code how to handle code of mixed sensitivity. Mitigations enforce a security policy for the code they protect. One could argue that mitigations therefore require higher privileges as they are critical to the security guarantees they provide. Compilers are unaware of this special nature of mitigations and may break inherent assumptions of the mitigation during optimization.

As we have discussed in § 2.4, bound-checked indirect jumps are secure as long as the checked value and the value to calculate the jump address is consistent. However, the consistence can be vulnerable to an attacker when the compiler introduces a double-fetch: the first time for the bound check and the second time for the jump address calculation.

4.1. Vulnerable Double-Fetch

When double-fetched, the checked value of the bound-checked indirect jump is vulnerable. That is because then there will be a TOCTTOU issue, i.e., it will leave a time window for the attacker to rewrite the checked value after the bound check and control the target of the indirect jump.

Definition of Vulnerable Double-Fetch. Two memory reads are vulnerable to our double-fetch attack if they satisfy the following conditions:

- The first memory read targets a bounds check, e.g., implemented through a branch. If satisfied, execution continues to an indirect control-flow transfer, e.g., through a jump table. If not satisfied, the default case will be triggered.
- The second memory read (of the same address) determines the target address of the control-flow transfer, e.g., by specifying the index into the jump table and loading the target address from the table.
- These two memory reads should be performed on the same value. Note that the two reads do not need to be performed on the same memory address. For example, the second read can be performed on the memory object where the value of the first read is spilled.
- The memory object of the second read should be writable.

Variant 1: fetch-fetch. The compiler generates code that fetches the checked value directly from memory for two times. As shown in Figure 3, the binary code fetches `[r13 + 0x118]` the first time at line 11 to do the check and then fetches it another time at line 15. The second fetched value is used to calculate (line 17) the indirect jump at line 19.

Variant 2: fetch-spill-fetch. The compiler sometimes will produce code that first fetches the value from memory to do the check, then spills the value to another place on the stack, and finally loads it from the stack to calculate the jump target. As shown in Figure 4, the binary code fetches the value to check at line 3, then the value is spilled on stack (`[rbp - 0x150]`) at line 4. The first fetched value is checked at line 5. Then the code fetches the spilled value at line 7 to calculate the jump target of the indirect jump at line 11.

Why it hurts code reuse defense. Compilers introducing such double-fetches open loopholes for code reuse defenses such as CFI that validate control-flow transfers. These vulnerabilities are caused by a misalignment between CFI

```
;switch (mca_p->port_format) {
;case SDP_PORT_NUM_ONLY:
;     SDP_PRINT("Port num %d, ", mca_p->port);
;     break;
;...
;default:
;     SDP_PRINT("Port format not valid, ");
;     break;
;}

 cmp dword [r13 + 0x118], 6
 pop rcx
 pop rsi
 ja 0x9d6720
 mov eax, dword [r13 + 0x118]
 lea rdx, [0x04fee16c]
 movsxd rax, dword [rdx + rax*4]
 add rax, rdx
 jmp rax
```

Figure 3. An example compiler-introduced double-fetch of a switch jump table

```
mov byte [rbp - 0x12d], al
mov qword [rbp - 0x138], rsi
movzx eax, byte [rbp - 0x12d]
mov qword [rbp - 0x150], rax
sub rax, 4
ja 0x80ac3d
mov rax, qword [rbp - 0x150]
lea rcx, [0x00233d40]
movsxd rax, dword [rcx + rax*4]
add rax, rcx
jmp rax
```

Figure 4. A variant of compiler-introduced double-fetch of a switch jump table

mechanisms and compiler optimizations. As we have discussed above, the compiler-introduced double-fetch leaves a time window for the attacker to modify the memory and corrupt the target of the indirect jump. Specifically, such code can be exploited in the attack model of a CFI and no CFI provides customized protection for such situation.

5. Attack Methodology

This section presents our methodology to perform attacks against applications protected with CFIs. It begins with a high-level demonstration of how the attack works (§ 5.1), then dive into the details by showing the implementation of a Proof-of-Concept attack against Firefox (§ 5.2). Note that our attack does not depend on any specific feature of LLVM-CFI and can bypass other CFI implementations as we demonstrate in the PoC example.

5.1. High-level Methodology

At a high level, our attack takes advantage of one type of compiler-introduced gadget code (double-fetch) that cannot be protected by any CFI implementation. This is a design weakness of CFI: the side effects of compiler transformations on the security assumptions of CFI are overlooked. With such gadget code and the same capabilities as outlined in the original CFI paper [10], the attacker can therefore hijack the control flow. The details of the adversarial capabilities of attackers are shown in § 3.

Specifically, our attack exploits the compiler-introduced double-fetch of a bound-checked indirect jump with a jump table. Regardless of compiler optimizations, the indirect jump is immune to code reuse attacks and does not need a CFI protection. However, as described in § 4.1, once the compiler introduces an extra fetch and uses the fetched value to calculate the jump target, the bound check of the index of the jump table can be bypassed by an attacker to gain control over the instruction pointer.

Figure 5 illustrates the timeline of the attack. We use x86/64 assembly-like pseudo code to represent the victim code. Note that real victim code may be more complex but should include all the components shown here.

1) The attacker rewrites the checked object to avoid leading the execution to the default branch; the attacker rewrites a controlled object to the address of a malicious target.
2) The victim thread fetches the index object from memory and performs the bounds check on it.
3) The attacker rewrites the object that will be fetched later. For Variant 1 (fetch-fetch), this is the same object of the checked object. For Variant 2 (fetch-spill-fetch), this is the spilled object.
4) The victim thread does the second fetch, reading the attacker-corrupted value.
5) The victim thread loads the address of the jump target. Normally, it should be loaded from the jump table. However, as the computation of the address is based on the corrupted value, the victim thread reads the indirect jump address from the controlled memory object. Finally, the victim thread executes the indirect jump and lands on the malicious target.

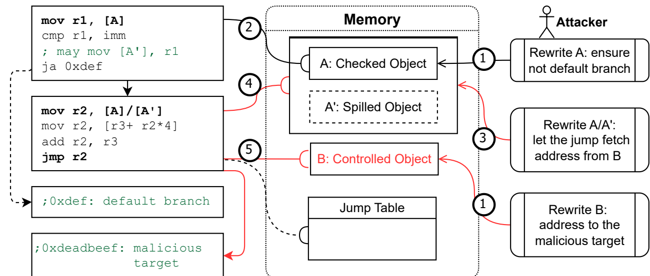

Figure 5. The timeline of exploiting the double-fetch to hijack control flow. `r1`,`r2`,`r3` means registers, `imm` means some random value and `[A]/[A']` means memory access of `A` or `A'`.

5.2. Proof-of-Concept Exploit

We demonstrate the effectiveness of our attack through a working exploit on a real-world program. For our proof-of-concept, we chose the Firefox Web browser because it is a complex and realistic target available for all common operating systems. We stress that our attacks also apply to other applications that provide the adversarial capabilities outlined in § 3.

In our PoC, we choose the last available version of Firefox at writing time (106.0.1) from the Fedora package manager, which is built by GCC (12.1.1). Our attack performs the following steps:

1) Gain arbitrary read/write capabilities.
2) Leak ASLR bases for both `libxul.so` and the stack.
3) Find double-fetch gadgets of bound-checked indirect jumps.
4) Find a way to reach one of the victim's vulnerable indirect jumps.
5) Orchestrate the thread scheduling to win the data race.
6) Overwrite the checked object and hijack the control flow.

Gaining arbitrary read and write capabilities. Attackers often obtain these capabilities from memory corruption vulnerabilities such as a use-after-free bugs [38] (e.g., CVE-2022-26485 for Firefox). We consider finding such vulnerability out-of-scope for this study, and assume the attacker can already read or write arbitrary addresses. Similar to previous work [18], we reintroduce a vulnerability in the browser. Specifically, we chose an out-of-bound bug that grants arbitrary read/write capabilities to the adversary through `ArrayBuffers` instances. The vulnerability is inspired from Capture-The-Flag competitions [39].

Gadget code detection. Based on the definition of vulnerable gadget code in §4.1, we implement a lightweight binary analysis tool based on Radare2 [40] to detect gadget code in a given binary. § 6.1 illustrates the specific pattern used to detect vulnerable gadgets.

Leak libxul.so base and stack address. The attacker requires to know the memory location of three objects: the address of the fetched object, the address of the victim jump table, and the address of one writeable memory region. The address of the fetched object is on the stack, the victim jump table is in the `.rodata` section of `libxul.so`. We then use the `.bss` section of libxul as a writeable section. To leak the necessary information, we rely on the `_elements` field of an `ArrayBuffer`, which contains a pointer to a predictable offset inside libxul. The pointer can be leaked through an out-of-bounds read from an adjacent `Uint8Array`, giving us the base address of `libxul.so`. With this information, we identify a double-pointer, called `__environ`, stored in the libc `.got` section and that points to the begin of the stack. By dereferencing those pointers, an attacker has all the leaks necessary for the exploit.

Reaching gadget code. The attacker just needs to reach one of the gadget code to attack successfully. While in our study we demonstrate that gadgets are common in real-world binaries (Table 2), they might be buried in complex logic. Therefore, finding a stable way to trigger one given gadget might be challenging (especially for a program as complex as a browser). We set up an experiment to find gadgets that can be easily triggered. Specifically, we add scripted breakpoints in GDB that log each time a vulnerable gadget is executed. We then use the browser like a regular user (e.g., navigating popular websites, watching videos). For browsers with hundreds of gadgets like Firefox, it took seconds to gather thousands of hits in the logs which we then analyze manually. We pick a Variant 1 type gadget in libxul's `TraceJitActivation()` which fires the JIT compilation when a given JavaScript *trace* has been executed frequently. This gadget is frequently activated at a fixed stack address, thus stabilizing our exploit. After some testing, we found out that a reliable way of triggering the vulnerable gadget was calling `document.getElementById('textarea').value += x` in a loop, modifying the value of `x` each time. Note that, for our PoC, we chose difficult-to-exploit gadget to demonstrate the feasibility of our attack in adverse conditions. More precisely, the chosen gadget shows the following characteristics:

- The TOCTTOU time window is small: only 2 instructions;
- If the overwriting happens too early inside the function containing the vulnerable jump table (i.e., before the double fetch gadget), the control flow will go into the default case and call `MOZ_CRASH()`, thus immediately crashing Firefox.

In real-world scenarios, adversaries may target more powerful gadgets to improve the success rate, e.g., gadgets with longer TOCTTOU time windows (such as those with a `jump` or `call` between fetches) or more robust to a failed overwrite (such as Variant 2 gadgets).

Thread scheduling. Since our attack exploits the inconsistency between the check and use sites of the compiler-introduced double-fetch, the attacker thread needs to overwrite the value of the checked value inside a certain time window. For Variant 1 (fetch-fetch), the time window is between the first fetch and the second fetch (between line 11 and line 15 in Figure 3); for Variant 2 (fetch-spill-fetch), the time window is between the spilling and the second fetch (between line 4 and line 7 in Figure 4). The time window can be as short as a few cycles. The most reliable way to win the race is constantly trying by putting both the victim thread and the attacker thread in a loop with different iteration speeds.

Overwriting. We note that, if the stack object address is disclosed correctly, multiple consecutive attempts of overwriting within one attack are usually possible, as failed trials incur no adverse changes (for Variant 2 gadgets) or lead the victim thread to the default branch of the switch statement (for Variant 1 gadgets). Furthermore, we propose a method to improve the robustness of failed overwriting. Overwriting a stack value while not executing the function containing the vulnerable jump table has a high probability to crash due to the corruption of unrelated stack variables. To increase the reliability of the attack, we design the attacker thread to check whether the return address of the function containing the gadget is present on the stack at each iteration. This approach prevents crashes due to overwriting unrelated local variables. As discussed above, an adversary can further mount an exploit based on Variant 2 gadgets to prevent negative side effects of losing the race. For example, Figure 4 the attacker overwrites the spilled object `[rbp -`

Algorithm 1 Measuring our PoC's success rate.

```
1: function EXPERIMENT
2:     while 2000 times do
3:         repeat                              ▷ one attempt
4:             attack()           ▷ run the race to overwrite
5:         until 20s have passed
6:     end while
7: end function
```

`0x150]` to win the data race; a failed attempt misses the correct time window but does not incur any other side effects as the spilled object is only used for calculating the indirect jump address in this function.

Bypassing Code-Reuse Defenses. Exploiting a compiler-introduced double-fetch to hijack the following indirect jump bypasses CFI. Specifically, the exploit does not require customization to bypass CFI as vulnerable CFI implementations do not protect these code patterns (as they assume that the indirect jump is protected through the bounds check). In other terms, WarpAttack works regardless of CFI because we exploit a misalignment between compiler optimizations and CFI implementations.

Success rate. We conduct experiments with our PoC on Firefox to test the success rate of the attack in a real-world scenario. We investigate the reliability of our attack by measuring the success rate with different combinations of attacking threads and dedicated cores. Specifically, we try 1, 3, and 7 attacker threads while running Firefox on 1, 4, and 8 cores, respectively. We conduct the experiments on an Intel(R) Core(TM) i7-10700 CPU (8 cores) @ 2.90GHz with 32GB of memory and Fedora 36. Algorithm 1 illustrates the pseudocode of the experiment, which is executed with each combination of thread and cores. The experiment is composed of two nested loops, the outer loop iterates $2,000$ times, while the inner loop executes the malicious data race for 20 seconds. The inner loop represents an attempt and can have three outcomes: success (i.e., we hijack the browser execution), failed without crash, and failed with crash. Then, we count the success rate as the number of success over $2,000$. Table 1 shows the results of our experiments: we observe the best success rate of 0.45% with 7 threads and 8 cores. This result is expected since increasing the number of attacker threads also increases the likelihood of success with sufficient cores. Conversely, with only one thread/core, the probability drops, i.e., we did not observe successful attacks with only one core. We further investigate the interference of multiple cores and threads in an ideal thread scheduling (Appendix A.3).

6. Gadget Code Detection

We develop a binary analysis tool to detect vulnerable gadgets in programs. We design our analysis for offensive purposes, i.e., without guarantee of soundness. For the adversary perspective, as long there are useful gadgets for an attack, and the false positive rate is relatively low (i.e., enough to find useful gadgets without manually reversing the whole program), then the results are useful for mounting an attack.

In the rest of the section, we describe the code pattern that allows our attack (§ 6.1). Then, we introduce our lightweight approach to detect such vulnerable code quickly (§ 6.2). Finally, we explain how our approach can work with low false negative (§ 6.3).

TABLE 1. DIFFERENT SUCCESS RATES BY TUNING NUMBER OF CORES AND NUMBER OF ATTACKER THREADS (IN 2000 RUNS).

#Core	#Attacker Threads		
	1	3	7
1	0	0	0
4	0.05%	0.25%	0.2%
8	0.15%	0.15%	0.45%

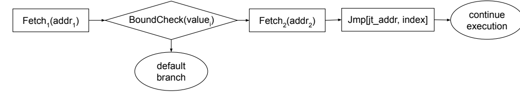

Figure 6. Simplified schema of vulnerable code pattern.

6.1. Vulnerable Code Pattern

Based on the definition of vulnerable double-fetch (§ 4.1), We describe the binary pattern as a combination of necessary vulnerable elements and the constraints. Figure 6 shows a pictorial representation of these patterns that we use in our analysis.

The vulnerable code needs to contain the following elements:

- **Two memory reads.** We define this requirement as $\texttt{Fetch}_1\texttt{(addr}_1\texttt{)}$ and $\texttt{Fetch}_2\texttt{(addr}_2\texttt{)}$. The two operations load a value from memory address $\texttt{addr}_1$ and $\texttt{addr}_2$, respectively. Moreover, we assume the two operations read the same amount of memory. In case they read different memory size, the larger chunk can always be treated as the smaller one.
- **Bound checks.** Defined as $\texttt{BoundCheck(value}_i\texttt{)}$, $\texttt{value}_i$ indicates the value being checked.
- **Indirect jumps based on jump tables.** We define it as `Jmp[jt_addr, index]`. The jump table is stored at `jt_addr` address, while `index` is the index of the table.

Given the above elements, they have to satisfy the following constraints to be considered vulnerable.

- **Control dependency.** The elements should appear in the order described in Figure 6. Here, $\texttt{BoundCheck(value}_i\texttt{)}$ decides whether the indirect jump should be taken, i.e., by executing $\texttt{Fetch}_2\texttt{(addr}_2\texttt{)}$ and `Jmp[jt_addr, index]`. If the bound check fails, the switch statement transfers control to the default branch.
- **Data dependency between $\texttt{Fetch}_1$ and BoundCheck.** The value of $\texttt{Fetch}_1\texttt{(addr}_1\texttt{)}$

should populate `value`$_i$. We allow non-direct assignments, e.g., involving calculations or being used as arguments for function calls.
- **Data dependency between the two fetch operations.** The value extracted by `Fetch`$_1$`(addr`$_1$`)` and `Fetch`$_2$`(addr`$_2$`)` must be identical. However, we allow `addr`$_1$ and `addr`$_2$ not to be the same. For instance, the two memory locations are populated from the same source, or else, the value from addr_1 is copied into addr_2. This situation happens in memory spilling when the first value is spilled into the second one.
- **Data dependency between `Fetch`$_2$ and `Jmp[]`.** The value of `index` (for the jump table) should depend on `Fetch`$_2$`(addr`$_2$`)`. This allows an arbitrary memory write to tamper with the content of `addr`$_2$ for controlling the value of `index` (as per § 3).

6.2. Our detection method

Upon the code pattern description in § 6.1, we propose a lightweight binary analysis to detect gadget code in x86/64 binaries with negligible false negative rate and low false positive rate (around 3% in our experiments § 7.1). Ideally, completely checking vulnerable code pattern requires a heavyweight data-flow analysis that could be infeasible in practice. As an alternative, we decide to check the constraints described in § 6.1 as a more practical solution. Specifically, we propose a set of heuristics for x86/64 architecture, however, they can be adapted for other architectures as well.

Our binary analysis tool leverages Radare2 to detect gadget code in a given binary program. Our key idea is to first greedily locate all the indirect jumps and then progressively match constraints to filter out false positives.

For a snippet of code to be classified as such, we require it fulfills all the following conditions:

C1 Jump table access: We select all the indirect jumps that follow this pattern: (1) the target address is calculated as a sum of a *base address* and an *offset*, (2) the *base address* is the result of an additional offset operation. For example, we refer to the last three lines in Figure 3 and Figure 4.

C2 Double fetches: First, we select pair memory accesses by checking a fixed interval of instructions (e.g., 10). Then, we check the second access is a read, while the first access can be either read or write. This check allows us to select double-fetches from the same address (Variant 1) and different addresses (Variant 2) § 4.1.

C3 Constraints between the first memory access and the bound check: We check if there are comparison instructions between the two memory accesses as extracted from **C2**. More precisely, we consider each x86/64 instruction that can act as an upper bound check, e.g., `cmp`, `sub`, `test`, `and`, `xor`, `dec`.

C4 Constraints between the second fetch and the indirect jump: The register used in the indirect jump (**C1**) should be used at least once between the second fetch and the computation of jump table address. Moreover, the use can not be a comparison.

6.3. Imprecision Discussion of our Analysis

Given the conditions in §6.2, we discuss possible sources of imprecision.

- **C1** may generate false negatives if the compiler employs different patterns for computing jump table access. In our evaluation, we verify **C1** holds for GCC, Clang, and MSVC. We therefore consider false negatives of this kind rare.
- **C2** may generate two kinds of false negatives. First, the instruction interval of the analyzer may be too short. In our evaluation, we observe an interval of 10 instructions is enough to cover the two memory accesses, the bound-checked, and the jump table access. Second, if we target at memory access pairs pointing the same address, we may miss cases where the second fetch holds the same value being checked, but the memory is in a different address and not directly assigned. In our experiments, we never encountered these cases, so we consider them rare.
- **C3** and **C4** may incur false positives since these checks are only approximation of formal constraints. Specifically, **C3** does not imply data dependency between the memory access and the bound check; **C4** only requires a necessary but not sufficient data dependency between the second fetch and the indirect jump, which does not guarantee the jump correctly uses the fetched value. Therefore, we consider **C3** and **C4** as conditions necessary but not sufficient for implying data dependency.

In practice, according to our evaluation (§ 7.1) our tool may (rarely) incur false negatives (we tune it to reduce this probability). Most importantly, it introduces a low false positives rate of 3% thus making our analysis suitable for offensive purposes. Appendix A.2 gives examples of false positives.

7. Evaluation

The real-world impact of our attack is mostly influenced by how many such compiler-introduced double-fetches gadgets in the wild. In this section, we look into this question to evaluate the real impacts of our attack. First, we study the existence of gadget code in widely used programs on mainstream platforms. Then we look into the affected compilers and CFI implementations. Finally, we give a brief evaluation of affected architectures. In general, we try to answer *four* research questions:

- RQ1: Do compiler-introduced double-fetch gadgets exist in real programs? (§ 7.1)
- RQ2: Which compiler is affected by such situation? (§ 7.2)
- RQ3: Which CFI implementation is vulnerable to our attack? (§ 7.3)
- RQ4: What architectures are affected by WarpAttack? (§ 7.4)

7.1. RQ1: Victim Code in the Wild

To estimate the impact of WarpAttack in real scenarios, we apply our binary analysis tool from § 6 to real-world programs of mainstream platforms. We select the test programs from the CFI compatibility metrics of CONFIRM [41]. We choose *six* cross-platform programs from the metrics and get the latest available pre-compiled version from the official websites and the package manager of each platform. Additionally, we select any of these programs for *five* OSs: Fedora, Debian, Ubuntu, Windows, and Mac OS. Note we chose three Linux distributions as they have different package managers. Sometimes Linux distributions maintain the same open-source program compiled with different compilers. Since our gadget analysis works for x86/64 architectures, we download the programs accordingly. In detail, we choose Chrome (103.0.5060), Firefox (103.0.2), Apache (2.4.54, not available for Mac OS), JVM (OpenJDK) (18.0.2), 7-zip (22.01, 21.07 for Mac OS), and TexStudio (4.3.0).

To validate our analysis, we manually double-check the results obtained. Due to the large amount of gadgets, we adopted this procedure. To reduce the probability of false negatives, we try different size for the instruction windows, from 0 to 20. We observe that, for the Intel architecture, a size of 10 instructions returns all the interesting indirect jumps: a larger window does not change the result, while a narrower window excludes real gadgets. Therefore, we are confident a window of 10 instruction is a good trade off between false positives and false negatives. To analyze false positives, we first cluster the remaining gadgets that share exactly the same instructions in the same order. Then, we manually sample from each cluster a subset of gadgets and manually validated the correctness. All in all, we observe around 3% of false positives among all the combination of programs/OSs. Appendix A.2 gives examples of false positives.

Table 2 shows the result of our analysis. Overall, we observe the combination of Chrome and Fedora contains the higher number of gadgets (1024). However, the gadgets for Chrome drops for the other OSs, ranging between 16 and 24. In general, programs belonging to Fedora contain more gadgets respect to other systems. Looking at the other systems, we notice for Mac OS and Ubuntu the programs have one order of magnitude lesser gadgets, 66 and 103, respectively. In general, since the programs are delivered already compiled, we do not have control on the compiler or the options used, therefore, we cannot pinpoint the source of a gadget. We observe a remarkable gap among OSs and programs. In case of complex projects, such as Chrome and Firefox, we assume they introduce more gadgets (e.g., due to their JS engine). As we will better discuss in § 7.2, the compiler that statistically inject more gadgets is GCC, which is the default compiler for Fedora, while Windows and Mac OS use MSVC and Clang as default compiler, respectively. This justifies the presence of more gadgets in Fedora. However, these projects are built with many pre-build libraries, that may be compiled from different sources with different options. Thus hindering a more accurate analysis.

TABLE 2. Statistics of double-fetch gadgets in the wild. We excluded Apache for Mac OS because we fail to find the correct pre-compiled version for Intel Mac OS.

Program	Fedora	Debian	Ubuntu	Windows	Mac OS
Chrome	1024	16	23	24	16
Firefox	616	659	31	0	29
Apache	15	17	16	0	-
JVM	0	0	0	0	1
7-zip	24	24	24	0	0
Texstudio	8	9	9	230	20
Total	1687	725	103	254	66

TABLE 3. Compilers that can introduce exploitable double-fetch paired to their compilation options. The symbol "*" indicates cases observed from real world programs.

Compiler	Option	double-fetch Type	version
GCC	`O1,O2,O3,Ofast`	Var. 1 (fetch-fetch)	12.1
*G++	`O1,O2,O3`	Var. 2 (fetch-spill-fetch)	12.1
Clang	`O0`	Var. 2 (fetch-spill-fetch)	14.0.*
*Clang	`O1,O2,O3`	Var. 2 (fetch-spill-fetch)	14.0.*
*Clang	`O3`	Var. 1 (fetch-fetch)	14.0.*
*Clang++	`O3`	Var. 1 (fetch-fetch)	14.0.*
MSVC	`Od`	Var. 1 (fetch-fetch)	19.32.*

Takeaway: Considering a false positive rate of 3% (from our manual analysis), we can roughly estimate from tens to thousands of gadgets for most of the pair program/OSs. This confirms that WarpAttack is a real threat present in many popular operating systems and common applications.

7.2. RQ2: A Study of Compilers

In this section, we study which compiler may introduce double-fetch vulnerabilities. Specifically, we focus on the three popular compilers used in the major operating systems: GCC, Clang and MSVC. These compilers are commonly used for C/C++ projects and are the base of many compiler-based CFI implementations.

Ideally, we would analyze the programs listed in Table 2. However, the programs are "stripped" and most of the debug information and build logs are unavailable, thus investigating the root cause of those gadgets is not always possible. Therefore, we follow two approaches. (1) For each compiler, we develop a set of case studies to make emerge double-fetch vulnerabilities, that we compile with their respective CFI protections enabled when available. (2) When possible, we interact with the developers.

Table 3 summarizes our finding: we conclude that all the three compilers introduce some form of double-fetch vulnerability if the correct optimization level is set. GCC introduces double-fetches Variant 1 with all the compiler level actives, but `O0`. Clang and MSVC, instead, seem to introduce this problem only with `O0`/`Od` (MSVC `Od` disables optimization, similar to `O0` for Clang/GCC). We suspect this

is the reason why Fedora contains more gadgets on average. Finally, our observations show Clang introduces double-fetch Variant 2 vulnerabilities. We leave further examples of double-fetch gadgets in Appendix A.1.

Double-fetch gadgets from real-world programs. From our interaction with developers, we observe non-trivial combinations of compiler options and code patterns that produce double-fetch gadgets. For example, in Firefox, Clang/Clang++ can introduce both Variant 1 (with `O3 DNDEBUG=1`) and Variant 2 (with `O1/2/3`). Moreover, G++ can introduce Variant 2 (with `O1/2/3`). These cases are less frequent but difficult to predict.

Takeaway: We demonstrate that modern compilers introduce double-fetch vulnerabilities for both Variant 1 and 2. As long as we do not find an automatic mechanism to reduce the number of double-fetches, setting the correct optimization level becomes crucial to reduce the likelihood of introducing double-fetch gadgets.

7.3. RQ3: Vulnerable CFI implementations

We study which CFIs are affected by WarpAttack. For this evaluation, we select *six* CFIs in total: *four* CFIs compiler-based, *one* binary based, and *one* hardware assisted. Specifically, we choose RAP [42], and VTV [13] for GCC; LLVM-CFI [12] for Clang; CFG [43] for MSVC; Lockdown [20] as binary only; and Intel CET [44] as hardware CFI. When possible, we deploy the CFIs over the test-case from § 7.2, and manually inspect the code. Otherwise we reach the authors for technical clarifications. We selected the compiler-based CFIs as representative of default protections employed into compilers, while the binary and hardware CFIs show the extension of WarpAttack.

Table 4 shows the result of our experiment. In short, *four* (out of *six*) CFIs show double-fetch gadgets or cannot be directly used as mitigations. This is caused by the lack of compiler comprehensions between security-related and application code, thus allowing insecure optimization that undermines the security protection. In the following, we discuss the three CFIs we do not manage to use directly.

Lockdown and RAP are unavailable due to outdated prototypes or commercial products, respectively. Therefore, we contact the respective developers and clarify the protection of jump tables. For Lockdown, the authors confirm their work ignores jump tables since they are considered protected. Conversely, RAP authors include compilation flags to remove jump tables from the compilation pipeline. This setting was introduced as countermeasure against micro-architecture attacks such as Spectre [45] and Meltdown [46].

We further investigate Intel CET [44] and observe it technically can protect any indirect jump dispatch, e.g., jump tables. However, CET needs software support (e.g., compiler or CFI implementation) to select targets to protect (i.e., inserting `ENDBRANCH` tags). Recalling § 2.4, jump tables are currently assumed protected, thus ignored from protection. Although one could over-extend `ENDBRANCH` to each indirect jump, this would increase the valid targets set, thus increasing the attack surface. Therefore, we consider Intel CET not a straightforward solution for our attack.

TABLE 4. CFI IMPLEMENTATIONS VULNERABLE TO OUR ATTACK.

CFI Type	Compiler	Vulnerable CFI
Compiler-based CFI	GCC	VTV [13]
	Clang	LLVM-CFI [12]
	MSVC	CFG [43]
Binary only CFI	–	Lockdown [20]

Takeaway: Most CFIs are vulnerable to the WarpAttack attack. From our research, only RAP disables jump tables to mitigate some micro-architectural attacks. Besides this particular instance, our study confirms that popular code-reuse defenses are vulnerable to our attack.

7.4. RQ4: Vulnerable Architectures

The compiler behavior of generating jump tables for switch-like semantics and reloading data from memory (including spilling register values and reloading it from memory later) is architecture dependent. The aforementioned patterns of compiler-introduced double-fetches (as in § 4.1) target x86 and x86-64. In this section we analyze: *do compilers generate double-fetches vulnerabilities of bound-checked indirect jump in other architectures?* We answer this question by enumerating other well-known target architectures.

We validate our hypothesis by prepared C test cases with `switch` statements having more than 10 branches. Then, we cross-compile the test cases with different compilers and for different architectures. We target x86, arm, RISCV, and MIPS, both 32 and 64-bit versions. The results are summarized in Table 5. First, we observe that double-fetch optimizations do not appear only in x86/64 architectures. Moreover, we notice compilers perform register spilling more frequently on arm and RISCV architectures. We suspect this is caused by the number of registers available, which forces the compiler to resort to register spilling.

As an example, we demonstrate vulnerable gadget code compiled by Clang `O0` on ARMv7. As in Figure 7, the bound check is performed on line 5 on the value fetched at line 2. This value is spilled to the stack on line 4. The second fetch of the value happens on line 8 where the fetched value is later used as the jump table index to calculate the target address (line 13). The vulnerable indirect jump is on line 15.

Takeaway: Double-fetch vulnerabilities are not a prerogative of Intel architectures. Indeed, other architectures suffer from this problem as well. Moreover, double-fetch seems to be worsened by the number of registers available (that may cause spilling).

8. Related Work

WarpAttack falls into code-reuse attacks, however, it introduces significant differences with its predecessors. In

TABLE 5. CONFIRMED VULNERABLE ARCHITECTURES AND INVOLVED VARIANTS AND COMPILERS.

	Variant 1 (fetch-fetch)	Variant 2 (fetch-spill-fetch)
X86/-64	GCC O1/O2/O3	Clang O0; MSVC Od
ARM 32/64	-	Clang O0; MSVC Od
RISCV 32/64	-	Clang O0
MIPS 32/64	GCC O1/O2/O3	-

```
1  ldr     x8, [sp, #16]
2  ldr     w8, [x8]
3  subs    w8, w8, #1
4  str     x8, [sp, #8] ;8-byte Folded Spill
5  subs    x8, x8, #17
6  cset    w8, hi
7  tbnz    w8, #0, .LBB2_20
8  ldr     x11, [sp, #8] ;8-byte Folded Reload
9  adrp    x10, .LJTI2_0
10 add     x10, x10, :lo12:.LJTI2_0
11 .Ltmp7:
12 adr     x8, .Ltmp7
13 ldrsw   x9, [x10, x11, lsl #2]
14 add     x8, x8, x9
15 br      x8
```

Figure 7. A double-fetch (Variant 2: fetch-spill-fetch) of a switch jump table introduced by Clang O0 on ARMv7.

this section, we compare WarpAttack with previous similar attacks and discuss their mitigations.

Code-reuse attacks. Since the seminal work of Shacham et al. on return-oriented programming [6], the community developed an array of more advanced attacks [5, 7–9, 26, 47]. Even though these attacks partially share our threat model (§ 3), they only focus on the CFI policy and assume their implementation is correct. Conversely, WarpAttack shows that the compiler may introduce implementation flaws (TOCTTOU) that allow one to bypass the protection while undetected by the policy.

Mitigations. One protection against code-reuse attacks is to enforce the correct execution flow of a process. To achieve this, many CFI-like protections have been proposed [48–53]. These mitigations impose stricter policies to enforce more precise CFI protections, such as for indirect jumps and virtual tables. However, they mainly focus on the policy themselves, without considering compilers side effects. Recent works for kernel protection [54] mentions possible double-fetch issues, admitting though they cannot prevent the threat. Following this line, our study shows double-fetch can bypass these mitigations.

Data-only attacks. With the advance of CFI policies, adversaries investigated more advanced attacks, called data-only, that bend the program logic without violating the execution-flow [30, 55, 56]. In principle, data-only attacks assume arbitrary read/write vulnerabilities to manipulate sensible non-control data. These attacks proof it is possible to carry attacks without breaking the CFI policies. Our attack, instead, focuses on hijacking the execution flow, thus aiming at different goals.

Double-fetch. Since double-fetch issues can lead to TOCTTOU attacks, the community deeply studied the problem from different angles [19, 25, 32, 33]. Regardless the different scenarios, all the previous works share different assumptions with our work. Specifically, they consider pure concurrency issues in complex systems like OS kernels, while ruling out attackers able to overwrite memory regions (neither from one nor multiple threads). Nearly all of our gadget code can not be exploited in their threat model and all these detections [32, 33, 57] and mitigations [19, 57] do not consider our gadget patterns and cannot be used to find our gadgets nor to prevent our attack. The closest work was discussed by Conti et at. [18], in which they exploit register spilling for leaking the stack base address. Conversely, WarpAttack employs a new technique based on TOCTTOU to bypass any CFI policies.

9. Discussion

In this section, we discuss the implication of our attacks (§9.1), and introduce possible mitigations (§9.2). Finally, we illustrate other possible compiler-introduce gadgets (§ 9.3), and general CFI weakness (§ 9.4);

9.1. Making compilers aware of sensitive code

The correctness of compilers has been extensively tested [58, 59], verified, and certified [60, 61]. However, a correct compiler can still produce insecure code, as the security attributes are often beyond correctness. D'Silva et al. [62] introduce the concept of correctness-security gap, in which a formally sound and correctly implemented compiler optimization can violate security guarantees incorporated in source code. A common case is when compilers apply "Dead Store Elimination" optimization to remove the sensitive data scrubbing (meaningless value usually appended after sensitive data, like a secret key) [63].

Things get to be more complicated when security mechanisms are involved. Some security mechanisms usually assume and require specific security guarantees to be held during the compilation in both the original source code and newly-inserted code. For instance, our attack relies on the fact that compilers are not aware of security attributes in code (in the perspective of code reuse defense). Therefore, the optimization may weaken their protection and breaking the underlying security assumptions.

A fundamental solution of such correctness-security gap would be making compilers aware of security sensitive code. For example, compilers may introduce different levels of code, where the sensitive code, e.g., CFI policy checks, bound checks of indirect jump, should undergo different optimizations based on the attacker model. There are some secure compilation works [64–66] that aim at preserving general security properties, such as confidentiality of values or side channel side effects [67]. However, the impact of compilers on security mechanisms has not been discussed in this area.

9.2. Mitigation Options

Since current compilers do not prevent double-fetch vulnerabilities yet. We discuss mitigations that can be currently adopted and their cost in terms of performances.

Avoiding Gadget code generation. The most direct way to mitigate our attack is to prevent compilers from producing gadget code. GCC provides the option `-fno-switch-tables` to avoid producing jump tables. This mechanism also reduces Spectre side channels and is already used in the Linux kernel [68]. For vulnerable programs compiled by Clang or MSVC, a simple workaround is to compile by using a higher optimization level than `O0`. However, we argue this solution may be unreliable, as GCC shows us the compiler updates may introduce such a double-fetch in later compiler versions. Therefore, we would suggest introducing a specific compilation check option to avoid double-fetch vulnerabilities.

Protecting Indirect Jump. Another promising mitigation is to add dynamic checks for every indirect jump including those related to a switch jump table. This would provide protection for potential unknown risks, even though it will incur performance overhead. Since CFI designs do not take compiler optimizations into consideration and compilers are constantly updating, it is possible that compilers may break the bound check of the switch indirect jumps in unpredictable ways. For example, MSCV eliminates switch bound checks if the default branch is unreachable.

Monitoring for Attack Behavior. WarpAttack's attacker thread exhibits some characteristics (e.g., spawning several threads, constantly writing a certain memory site) that signal if the target is under attack. Similarly, crashes (due to the low success rate) may signal an attack. But crashes are coarse-grained signals. First, benign bugs also crash the browser, resulting in false positives. Similarly, adversaries may adopt strategies to reduce (or even prohibit) crashes, e.g., checking the victim stack content (return address) to ensure the modification happens in the correct location, and using Variant 2 gadgets to prevent failed races from incorrectly directing control-flow to the default branch.

9.3. Other Compiler-Introduced Gadget

Besides double-fetch issues, we describe other types of gadgets introduced by a compiler. Specifically, compilers can eliminate `switch` bound checks when the default branch is unreachable. For instance, in the code compiled by MSVC in Figure 9, the checked value `x` is stored in `rcx` at the beginning. It is decremented and moved to `eax` at line 9 to clear the high bits and moved back to `rcx` at line 10. Then, the value is used without bound checks to calculate the indirect jump address at line 12 and 13. This gadget code can bypass a CFI. Because the absence of a bound check, the time window of this kind of gadgets for attackers is often longer than double-fetch ones.

We observe common compilers (e.g., GCC, Clang, MSVC) are vulnerable to this threat. The Windows CFI

```
1 mov dword [rsp + 0x18], 1
2 ...
3 mov eax, dword [rsp + 0x18]  ;the only fetch
4 lea rcx, [0x06e4e5b8]
5 movsxd rax, dword [rcx + rax*4]
6 add rax, rcx
7 jmp rax
```

Figure 8. One example in Firefox 103.0.2 where the bound check of the indirect jump is eliminated.

implementation (CFG [15]) integrates some compilation time checks for preventing the bound check elimination, as shown in Figure 9. Here, the binary compiled with such mitigation still holds the bound check at line 18. However, based on our observation, not all compilers/CFIs employ similar mitigation (e.g., LLVM-CFI).

In general, the attack method of WarpAttack can be applied to other indirect jumps whose target address is calculated based on a controllable memory object and are not protected by CFI instrumentation checks (or at least by some fine-grained bound-checks).

To estimate the magnitude of these cases, we implement a binary analysis tool to find code snippets satisfying the following constraints:

- they calculate indirect jumps;
- they involved non double-fetched memory objects;
- the jump address relies on a value from a controllable memory object. For simplicity, we only consider memory object accessed via `rbp` and `rsp`;
- there is no comparison between the memory load and address computation.

Similar to § 6, our analysis assumes an adversary who chooses the most suitable gadget for the exploit. We run the tool on the official x86-64 binary of Firefox 103.0.2 and get 120 reports. We then manually check and confirm that these indirect jumps may not be protected by CFI checks. Therefore, they can also be the target of WarpAttack. In addition to the double-fetch gadgets, our analysis finds two other cases: (i) gadgets using indirect jumps whose bound-checked is eliminated, (ii) indirect jumps without bound checks in Rust code.

Two of the reports are C++ cases and the remaining ones are in Rust code. The two C++ cases are bound-checked indirect jumps from `switch` statements or consecutive `if` statements whose bound checks are eliminated by compiler optimizations. The code of one case is shown in Figure 8: the bound check of the indirect jump is eliminated since the compiler statically infers the value from the `store` at line 1. This happens when code with bound-checked indirect jumps is inlined, and the checked value can be predicted statically. For the Rust cases, similar to other double-fetch cases, it is reasonable to assume they are not protected since Rust is a memory safe language. However, when paring Rust with legacy code (e.g., C/C++), an attacker can target the Rust gadgets too.

```
;switch (x) {
;    case 1:
;...
;    default:
;        __assume(0);
;}

; before CFG applying the prevention
lea      eax, DWORD PTR [rcx-1]
movsxd   rcx, eax
lea      rdx, OFFSET FLAT:__ImageBase
mov      eax, DWORD PTR $LN14@bar[rdx+rcx*4]
add      rax, rdx
jmp      rax

; after CFG applying the prevention
dec      ecx
cmp      ecx, 7
ja       SHORT $LN12@bar
movsxd   rax, ecx
lea      rdx, OFFSET FLAT:__ImageBase
mov      ecx, DWORD PTR $LN14@bar[rdx+rax*4]
add      rcx, rdx
jmp      rcx
```

Figure 9. An example where MSVC eliminates the bound check of the switch statement when the default branch is unreachable.

9.4. Other CFI weakness

The security community addressed threats introduced by compiler optimization. For example, CONFIRM [41] and Nathan et al. [37] show the implementation of ShadowStack on x86 may leave room for TOCTTOU attacks of return value (the time window is one-instruction long). However, this problem affects only some shadow stack implementation for x86/64. We argue that it can be prevented by translating a call into a `push jump` without the TOCTTOU issue.

10. Disclosure Process

We disclose our finding to the Firefox/Chromium team, the LLVM and Linux kernel community, Microsoft, and RAP/Grsecurity. Mozilla acknowledges the problem and helped us triage the compiler configuration in the pre-build versions. Through their interaction, we identify new configurations of Clang/Clang++ and G++ that generate double-fetch gadgets (§7.2). The Chromium developer team acknowledges WarpAttack and assigns a medium level of criticality. At the writing time, we are discussing with the Chromium team to triage the cause and design mitigations. The Linux Kernel team confirms kCFI [14] does not protect jump tables since they are considered out of the attacker range. Therefore, kCFI is affected by WarpAttack. However, the Linux kernel disables jump tables as protection against Spectre [45] and Meltdown [46]. Similar to the Linux Kernel, Grsecurity also includes compiler options to remove jump tables in both kernel- and user-space software as Spectre/Meltdown mitigation, thus intrinsically protecting RAP against WarpAttack (§ 7.3). Microsoft acknowledged the problem and we are currently discussing mitigations. Finally, we are interacting with LLVM to triage the cause of our gadgets and design mitigations.

11. Conclusion

In this work, we present WarpAttack, a novel double-fetch attack that undermines any control-flow protection. Our attack is based on compiler optimizations that break the CFI assumptions, thus leaving room for TOCTTOU attacks and finally granting arbitrary code execution to adversaries.

We demonstrate the feasibility of our attack with a proof-of-concept exploits against Firefox (version 106.0.1). Moreover, we developed a lightweight static analysis (with 3% of false positives) to locate vulnerable gadgets in popular programs, showing this threat is widespread across programs and OSs as well. We further investigate the presence of double-fetch issues across different architectures and CFIs. Our findings confirm that WarpAttack is a real threat, that should be taken into consideration by the community.

Finally, we linked the cause of WarpAttack to the correctness-security gap highlighted by D'Silva et al. [62]: compilers do not distinguish between security and application code. Therefore, they apply the same optimizations strategies without reasoning about the security implications. To mitigate WarpAttack, we suggest annotating security-related code, so that compilers can treat it accordingly.

In summary, WarpAttack is an emerging threat that opens new discussions in the security and compiler communities about the security implications of certain code optimizations.

Acknowledgement

We thank the anonymous reviewers for their feedback. This work was supported, in part, by grants from SNSF PCEGP2_186974, DARPA HR001119S0089-AMP-FP-034, AFRL FA8655-20-1-7048, ERC StG 850868, the Chinese National Natural Science Foundation (62032010,62172201), China Scholarship Council and Postgraduate Research & Practice Innovation Program of Jiangsu Province. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.
[2] T. PaX, "Pax address space layout randomization (aslr)," *http://pax.grsecurity.net/docs/aslr.txt*, 2003.
[3] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.
[4] Microsoft, "Data execution prevention (dep)," 2006.
[5] R. Wojtczuk, "The advanced return-into-lib (c) exploits: Pax case study," *Phrack Magazine, Volume 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e*, vol. 70, 2001.

[6] H. Shacham, “The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86),” in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007, pp. 552–561.

[7] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 559–572.

[8] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “Jump-oriented programming: a new class of code-reuse attack,” in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011, pp. 30–40.

[9] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, “On the expressiveness of return-into-libc attacks,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 121–141.

[10] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, “Control-flow integrity,” in *Proceedings of the 12th ACM conference on Computer and communications security*, 2005, pp. 340–353.

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, “Control-flow integrity principles, implementations, and applications,” *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[12] L. team, “Clang 16.0.0git documentation: Control flow integrity,” *https://clang.llvm.org/docs/ControlFlowIntegrity.html*, 2022.

[13] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing {Forward-Edge}{Control-Flow} integrity in {GCC} & {LLVM},” in *23rd USENIX security symposium (USENIX security 14)*, 2014, pp. 941–955.

[14] A. Docs, “Kernel control flow integrity,” *https://source.android.com/devices/tech/debug/kcfi*, 2022.

[15] M. Docs, “Control flow guard for platform security,” 2022, https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard.

[16] S. Mergendahl, N. Burow, and H. Okhravi, “Cross-language attacks,” in *Proceedings 2022 Network and Distributed System Security Symposium. NDSS*, vol. 22, 2022, pp. 1–17.

[17] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, “Control-flow integrity: Precision, security, and performance,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 1, pp. 1–33, 2017.

[18] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, “Losing control: On the effectiveness of control-flow integrity under stack attacks,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 952–963.

[19] A. Bhattacharyya, U. Tesic, and M. Payer, “Midas: Systematic kernel TOCTTOU protection,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 107–124. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/bhattacharyya

[20] M. Payer, A. Barresi, and T. R. Gross, “Lockdown: Dynamic control-flow integrity,” *arXiv preprint arXiv:1407.0549*, 2014.

[21] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, “Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 941–955. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice

[22] B. Zeng, G. Tan, and G. Morrisett, “Combining control-flow integrity and static analysis for efficient and validated data sandboxing,” in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 29–40. [Online]. Available: https://doi.org/10.1145/2046707.2046713

[23] B. Niu and G. Tan, “Modular control-flow integrity,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 577–587.

[24] F. Wilhelm, “Xen xsa 155: Double fetches in paravirtualized devices,” 2020, https://insinuator.net/2015/12/xen-xsa-155-double-fetches-in-paravirtualized-devices/.

[25] P. Wang, K. Lu, G. Li, and X. Zhou, “A survey of the double-fetch vulnerabilities,” *Concurrency and Computation: Practice and Experience*, vol. 30, no. 6, p. e4345, 2018.

[26] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications,” in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[27] T. H. Dang, P. Maniatis, and D. Wagner, “The performance cost of shadow stacks and stack canaries,” in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, 2015, pp. 555–566.

[28] A. Biondo, M. Conti, and D. Lain, “Back to the epilogue: Evading control flow guard via unaligned targets.” in *Ndss*, 2018.

[29] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, “Stitching the gadgets: On the ineffectiveness of {Coarse-Grained}{Control-Flow} integrity protection,” in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[30] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, “Block oriented programming: Automating

data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1868–1882. [Online]. Available: https://doi.org/10.1145/3243734.3243739
[31] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, "Enforcing unique code target property for control-flow integrity," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1470–1486.
[32] P. Wang, J. Krinke, K. Lu, G. Li, and S. Dodier-Lazaro, "How Double-Fetch situations turn into Double-Fetch vulnerabilities: A study of double fetches in the linux kernel," in *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/wang-pengfei
[33] M. Xu, C. Qian, K. Lu, M. Backes, and T. Kim, "Precise and scalable detection of double-fetch bugs in os kernels," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 661–678.
[34] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi, "Missing the point (er): On the effectiveness of code pointer integrity," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 781–796.
[35] S. Groß, "Exploiting a cross-mmap overflow in firefox," 2017, https://saelo.github.io/posts/firefox-script-loader-overflow.html.
[36] WHATWG, "Html living standard: Web workers," 2022, https://html.spec.whatwg.org/multipage/workers.html#workers.
[37] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 985–999.
[38] Y. Younan, "Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers." in *NDSS*, 2015.
[39] blazefox, "blazefox - blaze ctf 2018," https://devcraft.io/2018/04/27/blazefox-blaze-ctf-2018.html, 2018.
[40] R. Team, "Radare2 github repository," https://github.com/radare/radare2, 2017.
[41] X. Xu, M. Ghaffarinia, W. Wang, K. W. Hamlen, and Z. Lin, "{CONFIRM}: Evaluating compatibility and relevance of control-flow integrity protections for modern software," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1805–1821.
[42] P. Team, "Rap: Rip rop," in *Hackers 2 Hackers Conference (H2HC)*, 2015.
[43] M. CGH, "/guard (enable control flow guard)," https://docs.microsoft.com/en-us/cpp/build/reference/guard-enable-control-flow-guard?view=msvc-170, 2021.
[44] B. V. Patel, "A technical look at intel's control-flow enforcement technology," *Intel,[Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/technical-look-control-flow-enforcement-technology.html*, 2020.
[45] 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," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.
[46] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, and R. Strackx, "Meltdown: Reading kernel memory from user space," *Commun. ACM*, vol. 63, no. 6, p. 46–56, may 2020. [Online]. Available: https://doi.org/10.1145/3357033
[47] V. Kuznetzov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, *Code-Pointer Integrity*. Association for Computing Machinery and Morgan & Claypool, 2018, p. 81–116. [Online]. Available: https://doi.org/10.1145/3129743.3129748
[48] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, Aug. 2013, pp. 337–352. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang
[49] M. Payer, A. Barresi, and T. R. Gross, "Fine-grained control-flow integrity through binary hardening," in *Proceedings of the 12th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9148*, ser. DIMVA 2015. Berlin, Heidelberg: Springer-Verlag, 2015, p. 144–164. [Online]. Available: https://doi.org/10.1007/978-3-319-20550-2_8
[50] K. Lu and H. Hu, "Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 1867–1881. [Online]. Available: https://doi.org/10.1145/3319535.3354244
[51] R. Gawlik and T. Holz, "Towards automated integrity protection of c++ virtual function tables in binary programs," in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 396–405. [Online]. Available: https://doi.org/10.1145/2664243.2664249
[52] A. Prakash, X. Hu, and H. Yin, "vfguard: Strict protection for virtual function calls in cots c++ binaries." in *NDSS*, 2015.
[53] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in *NDSS*, 2015.
[54] S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, "In-Kernel Control-Flow integrity on commodity

OSes using ARM pointer authentication," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 89–106. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/yoo
[55] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-Flow bending: On the effectiveness of Control-Flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini
[56] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.
[57] M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard, "Automated detection, exploitation, and elimination of double-fetch bugs using modern cpu features," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, 2018, pp. 587–600.
[58] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011, pp. 283–294.
[59] V. Le, M. Afshari, and Z. Su, "Compiler validation via equivalence modulo inputs," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 216–226, 2014.
[60] M. A. Dave, "Compiler verification: a bibliography," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 6, pp. 2–2, 2003.
[61] J. McCarthy and J. Painter, "Correctness of a compiler for arithmetic expressions," *Mathematical Aspects of Computer Science*, vol. 1, 1967.
[62] V. D'Silva, M. Payer, and D. Song, "The correctness-security gap in compiler optimization," in *2015 IEEE Security and Privacy Workshops*. IEEE, 2015, pp. 73–87.
[63] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko, "Dead store elimination (still) considered harmful," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1025–1040.
[64] P. Agten, R. Strackx, B. Jacobs, and F. Piessens, "Secure compilation to modern processors," in *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, 2012, pp. 171–185.
[65] M. Patrignani and D. Garg, "Secure compilation and hyperproperty preservation," in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 2017, pp. 392–404.
[66] S. T. Vu, K. Heydemann, A. de Grandmaison, and A. Cohen, "Secure delivery of program properties through optimizing compilation," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 14–26. [Online]. Available: https://doi.org/10.1145/3377555.3377897
[67] M. Patrignani, A. Ahmed, and D. Clarke, "Formal approaches to secure compilation: A survey of fully abstract compilation and related work," *ACM Computing Surveys (CSUR)*, vol. 51, no. 6, pp. 1–36, 2019.
[68] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the spectre era," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1871–1885.

```
1 cmp DWORD PTR tv66[rsp], 17
2 ja  $LN22@do_op
3 movsxd  rax, DWORD PTR tv66[rsp]
4 lea rcx, OFFSET FLAT:__ImageBase
5 mov eax, DWORD PTR $LN24@do_op[rcx+rax*4]
6 add rax, rcx
7 jmp rax
```

Figure 10. Example of double-fetch Version 1 for MSVC `Od`. The first and the third line are the two fetches, respectively.

```
1 mov     eax,DWORD PTR [rdi+0x30]
2 add     eax,0xffffffff
3 cmp     eax,0x11
4 ja      40117e <do_op+0x3e>
5 add     rdi,0x30
6 jmp     QWORD PTR [rax*8+0x402010]
```

Figure 11. Example of correct jump table implementation for Clang `O2`.

Appendix A.

A.1. Examples of double-fetch

We list some examples of double-fetch vulnerabilities that we observed. Specifically, Figure 3 appears with GCC/`O2`, while Figure 4 is from Clang/`O0`. Finally, Figure 10 is an example of MSVC/`Od` where the first fetch is at line 1, and the second fetch at line 3. Additionally, we show some example where double-fetches do not appear: Figure 11, Figure 12, and Figure 13 are snippets from Clang/`O2`, GCC/`O0`, and MSVC/`O2`, respectively, in which there is only one fetch once at line 1. In all these cases, WarpAttack cannot work.

A.2. Gadget Detection in False Positive

We illustrate two representative examples of false positives observed with our Gadget Code Detection analysis (§ 6). In Figure 15, `[rbp - 0x38]` is double-fetched but the value is unrelated with `jmp rsi`. The second case, instead, is a disassemble errors of Radare2. As shown in Figure 16, the instruction `mov rcx, qword [r12 + 0xd8]` is confused with `add byte [rax], al`. In particular, this error seems to appear at the first or last instruction when disassembling narrow instruction windows.

```
1 mov     rax,QWORD PTR [rbp-0x8]
2 mov     eax,DWORD PTR [rax]
3 cmp     eax,0x12
4 ja      401320
5 mov     eax,eax
6 mov     rax,QWORD PTR [rax*8+0x402010]
```

Figure 12. Example of correct jump table implementation for GCC `O0`

```
1 mov eax, DWORD PTR sharedArr+44
2 dec eax
3 cmp eax, 17
4 ja  $LN24@main
5 lea rdx, OFFSET FLAT:__ImageBase
6 cdqe
7 mov ecx, DWORD PTR $LN62@main[rdx+rax*4]
8 add rcx, rdx
9 jmp rcx
```

Figure 13. Example of correct jump table implementation for MSVC `O2`

A.3. Multi-thread experiments

We crafted a toy example of our attack on custom victim code to test scheduling trade-offs if the race window is a single instruction. The success rate of the attack is 100% and the time delay is negligible, i.e., even a single instruction race can be won quickly. On average, it takes 0.04 seconds to win the race for the first time and successfully perform the attack; the victim thread executes on average 86,095 iterations, the attacker thread 14,689 iterations. Figure 17 shows our toy attack. We use the function `corrupted()` to simulate the attacker behavior, the function `victim()` to simulate a victim thread with vulnerable gadget, and the function `evil()` is the target of the control-flow hijack. The time window for attacker in the victim code is between line 3 and line 5. As shown in `main()`, we assume the attacker can get the ideal thread scheduling: the victim thread constantly loops (line 32), and the attacker controls its thread to repeat with a different loop speed at the same time. Table 6 shows different performances at varying of cores and attacker threads. The result shows that an high number of attack threads increases the attacking time. This is reasonable since the clock is a finite resources, thus increasing the number of running threads delays attack.

```
1 cmp byte [rsp + 0x10], 3
2 ja 0x2f9e1ef
3 mov eax, dword [rsp + 0x10]
4 lea rdx, [0x06e8fbd0]
5 movsxd rax, dword [rdx + rax*4]
6 add rax, rdx
7 jmp rax
```

Figure 14. Example of a Variant 1 gadget introduced by Clang `O3` in Firefox 103.0.2.

```
1 cmp qword [rbp - 0x38], 0    ; the first fetch
2 je 0x5ed5928
3 mov rdx, qword [rbp - 0x38]    ; the second fetch
4 mov edx, dword [rdx + 8]
5 dec esi
6 lea rdi, [0x01b34c98]
7 movsxd rsi, dword [rdi + rsi*4]
8 add rsi, rdi
9 jmp rsi
```

Figure 15. `[rbp - 0x38]` is double-fetched but unrelated with `jmp rsi`. This sample comes from `./opt/google/chrome/chrome` of `chrome-stable-103.0.5060.134`.

TABLE 6. TIME TO REACH FIRST SUCCESS (TOY EXAMPLE) BY TUNING NUMBER OF CORES AND NUMBER OF ATTACKER THREADS IN 2000 RUNS.

# Cores	# Att. Threads	Avg. Victim Iterations	Avg. Attacker Iterations	Avg. Time [ms]
2	1	1190	166	2
2	2	1024	3708	3
2	3	2258	21099	6
2	4	763	17134	13
2	7	1283	11293	8
2	8	697	3239	3
2	16	93726	363345	49
4	1	25302	3191	10
4	2	20227	15112	23
4	3	2885	9825	9
4	4	2781	10772	12
4	7	1743	10212	15
4	8	1590	10446	16
4	16	2293	27201	23
8	1	54052	9056	19
8	2	24428	22159	39
8	3	12980	31160	57
8	4	5871	19228	58
8	7	10288	63720	291
8	8	9191	70545	307
8	16	10709	186458	695

```
1 ; radare2 output
2 add byte [rax], al
3 xor eax, eax
4 ; correct decompilation
5 mov rcx, qword [r12 + 0xd8]
6 add r12, 0xd8
7 xor eax, eax
```

Figure 16. Radare2 erroneously disassemble the first instruction as `add byte [rax], al`, which does not correspond with the real bytecode.

```
// <victim>:
// ...
// cmp    QWORD PTR [rax+0x30],0x6; op->v
// ja     bc0 <victim+0x80>
// mov    rdx,QWORD PTR [rax+0x30]; op->v
// lea    rcx,[rip+0x233]
// movsxd rdx,DWORD PTR [rcx+rdx*4]
// add    rdx,rcx
// jmp    rdx
// ...

void* corrupted(void *controlled) {
...
 // we assume the object is controlled
 * (int*)controlled = FAKE_JT_OFFSET;
 for (;;) {
 *(volatile typeof(op->v) *)&(op->v) = 2;
 *(volatile typeof(op->v) *)&(op->v)
                            = FAKE_CASE;
 }
...
}

void evil() { printf("Welcome to evil\n"); }

int main(int argc, char *argv[]) {
...
 // we assume the object is controlled
 int* controlled =
            (int *)malloc(sizeof(int));
 // start corrupted threads
 for (i = 0; i < ATTACKER_NUM; i++) {
    rc = pthread_create(&threads[i], NULL,
            corrupted, (void *)controlled);
 }
 // start the victim thread
 for (;;) { victim();}
...
}
```

Figure 17. A toy example that simulates an exploit, mostly about thread scheduling.