

Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs

Jianhao Xu¹ Kangjie Lu² Zhengjie Du¹ Zhu Ding¹ Linke Li¹ Qiushi Wu² Mathias Payer³ Bing Mao¹

¹State Key Laboratory for Novel Software Technology, Nanjing University

²University of Minnesota

³EPFL

Abstract

Compilers assure that any produced optimized code is semantically equivalent to the original code. However, even “correct” compilers may introduce security bugs as security properties go beyond translation correctness. Security bugs introduced by such correct compiler behaviors can be disputable; compiler developers expect users to strictly follow language specifications and understand all assumptions, while compiler users may incorrectly assume that their code is secure. Such bugs are hard to find and prevent, especially when it is unclear whether they should be fixed on the compiler or user side. Nevertheless, these bugs are real and can be severe, thus should be studied carefully.

We perform a comprehensive study on compiler-introduced security bugs (CISB) and their root causes. We collect a large set of CISB in the wild by manually analyzing 4,827 potential bug reports of the most popular compilers (GCC and Clang), distilling them into a taxonomy of CISB. We further conduct a user study to understand how compiler users view compiler behaviors. Our study shows that compiler-introduced security bugs are common and may have serious security impacts. It is unrealistic to expect compiler users to understand and comply with compiler assumptions. For example, the “no-undefined-behavior” assumption has become a nightmare for users and a major cause of CISB.

1 Introduction

Compiled code must conform to the source code. The correctness of compilers is therefore tested [49, 118] and verified [27, 69]. A compiler is considered correct if it produces code that is semantically equivalent to the original [80]. Such a correctness assurance however does not ensure security. D’Silva et al. [30] presented multiple cases in which correctly implemented compiler optimizations still introduce severe security issues like information leaks. We refer to *compiler-introduced security bugs* as *CISB*.

Several fundamental reasons cause compilers to introduce security bugs. (1) The security-related program states exceed

the scope of semantic functionalities of language specifications. The abstraction of source code can cover program states related to security but not semantic functionalities, e.g., the lifetime/region of sensitive data. Correctly implemented compiler optimizations, however, cannot preserve such program states by design. Figure 1 shows an example in the Linux kernel; the `memset` at line 5 is supposed to scrub sensitive data on memory to prevent information leaks. However, without specification on how to prevent information leaks (which is orthogonal to functional semantics), compilers may eliminate this `memset` after inferring that the stored variable `hash` is never read later. As a countermeasure, developers must resort to `memzero_explicit` at line 6 which prohibits compiler optimization. (2) The specifications can be implicit. The specifications permit a compiler to provide the correctness assurance just for so-called “well-defined” code. However, program states outside the scope (such as some undefined behaviors) can also be used to represent security properties. These implicit specifications allow the compiler to do aggressive optimizations that may break security properties. For example, a programmer may add a bound check like `if(x+10<x)` to detect a signed integer overflow of `x`, but the compiler may assume (according to the language specification) that the source code is free of signed overflows and thus eliminates the check.

Such manipulation of security-related program states can be opaque. As a result, even experienced developers may inadvertently and unknowingly write code that can be optimized to contain security bugs. Existing approaches use tricks [29, 78]

```
1 /* commit: d4c5efdb97773f59a2b711754ca0953f24516739 */
2 /* drivers/char/random.c */
3 static void extract_buf(struct entropy_store *r, __u8 *out) {
4     ...
5     - memset(&hash, 0, sizeof(hash));
6     + memzero_explicit(&hash, sizeof(hash));
7 }
```

Figure 1: A Linux kernel patch for preventing the GCC elimination of sensitive-data scrubbing. Note that `memzero_explicit()` has been patched twice later: The first [91] prevented an optimization in GCC, while the second [74] targeted Clang.

to “tame” the compiler; however, these tricks may fail with a new compiler, as compiler optimizations constantly improve.

Because of these fundamental reasons, *CISB* are surprisingly common. For example, Wang et al. [109] uncovered 160 such bugs in widely deployed systems; Lu et al. [65] detected 13 known and 10 new compiler-introduced information-leak vulnerabilities in the Linux kernel. In addition to the findings of these targeted studies, many *CISB* are reported by end-users in the wild, such as a large number of CVE-assigned vulnerabilities [71, 72]. *CISB* can be exploitable, e.g., Jian et al. [90] convert a *CISB* which eliminates a null pointer check to a Use-After-Free, escaping the Chrome sandbox in the Tianfu Cup 2020. *CISB* must be found and fixed just like the bugs introduced by developers, but they are more challenging to discover (thus having a longer latent period), i.e., the number of undiscovered *CISB* may be large.

Therefore, it is important to study *CISB*. The research field is however under-studied (§10). D’Silva et al. [30] defined the *correctness-security gap* and three classes of such security bugs by reasoning about them. Wang et al. [108] collected a list of real-world examples of 7 known undefined behaviors. Wang et al. [109, 110] developed a static checker to detect *CISB* due to a dozen kinds of undefined behavior. Lu et al. [65] investigated information-leak bugs introduced by uninitialized padding bytes introduced by compilers. Yang et al. [119] and Sprundel [103] studied *CISB* caused by Dead Store Elimination. Yang et al. [119] developed an optimization pass for scrubbing-safe dead store elimination. These studies each focus on a few patterns, but the breadth of the problem calls for a comprehensive study of real-world *CISB*.

We first conduct a labor-intensive study to collect *CISB* in the wild. (§2) From the tremendous number of bug reports reported to both the compilers and target programs, we perform our filtering strategy and get a set of potential *CISB* reports with 4,827 cases. Then we spend about 1,500 person-hours analyzing, confirming, and reproducing all potential *CISB*.

Based on our *CISB* dataset, we further investigate the following research questions:

- RQ1: What are the causes, formation, and security impacts of *CISB*? (§3, §4, and §5)
- RQ2: What are the knowledge and views of programmers about *CISB* and the mitigations? (§6)
- RQ3: What are the risks of existing mitigations? (§7)
- RQ4: What are the challenges and opportunities for further research? (§8)

We derive multiple important findings and lessons from the study, including: (1) *CISB* are much more diverse than previously thought. (2) *CISB* are common in the wild and have severe security impacts. (3) The widely adopted “no-undefined-behavior” assumption for compiler optimizations has become a major cause of *CISB*. (4) A large percentage of C programmers are oblivious to this assumption, and there is still a knowledge gap between knowing about the assumption

and avoiding *CISB*. (5) The *CISB* mitigations provided by compilers are effective for a small part of *CISB* but suffer from high-performance overhead. (6) The *CISB* mitigations may further be bypassed by compiler optimizations.

Through this study, we highlight future research opportunities on secure compilation. In particular, compiler users are expected to prevent *CISB*, while they generally have less knowledge of *CISB* compared to compiler developers. Thus, *CISB* still widely exist. We should instead develop automated techniques that avoid problematic compiler optimizations or precisely detect potential victim code. This research will hopefully guide and motivate further research; secure compilation still has a long way to go. To this, we contribute the following:

- We identify a large set of different kinds of *CISB* in the real world with a practical methodology. Our *CISB* dataset is available at <https://sites.google.com/view/cisb-study>. We propose a taxonomy of *CISB* (Table 4) based on the root causes, formation, and security impacts for each class.
- We perform a user study (N=62) demonstrating that C programmers often lack knowledge and have difficulty understanding *CISB* (§6). The questionnaire is available in our published dataset. We hope the study results provide feedback to security and compiler research.
- We investigate and show the risks of existing mitigations. Specifically, (i) we show *CISB* prevention performed by programmers is risky, derived from real cases; (ii) we perform a comprehensive evaluation of existing mitigations provided by compilers, with our dataset.
- We shed light on future research based on the new knowledge obtained and revisit current research progress on this problem. By showing the challenges and opportunities ahead, we hope this study serves as a motivation and guidance for future research on secure compilation.

2 The *CISB* Dataset

2.1 Problem Scope and Attacker Model

We define a software bug as a *CISB* when:

- the code, when executed without optimization, has no security issues on the target machine;
- compiler optimizations modify the code during compilation, creating the vulnerability;
- the code should not contain any incorrect usage of language keywords¹;
- the compiler optimization is formally correct, i.e., the compiler does not violate any language specification.

¹C keywords like C attributes provide a way to annotate language constructs such as variables, functions, or types, which can convey information to help compiler optimizations.

It is worth noting that the term “introduced” in *CISB* is only used to describe that the bug is created during the compilation phase, but is not present in the source code. The term is not supposed to assign responsibility.

Attacker Model. Our definition of a *security bug* is *impact-driven*. That is, we define a bug as a security bug when it can cause security impacts (breaking integrity such as memory corruption, availability such as crashing, or confidentiality such as leaking information). Triggerability (i.e., how to reach a bug) is out of scope because we focus on bug types rather than concrete bug cases. When conditions are met, all *CISB* types we studied can be reachable in specific programs.

Since our determination of a *CISB* is *impact driven*, in the following, we discuss common security impacts we encountered during our study.

- Introducing invalid instructions such as division-by-0. Such invalid instructions would crash (breaking availability) the program on some architectures.
- Introducing memory errors, including out-of-bound access, use-after-free, and uninitialized uses. Memory errors are generally security-critical, so they are in the scope of *CISB*.
- Introducing resource exhaustion such as infinite loops and deadlocks, breaking availability.
- Introducing race conditions such as TOCTTOU [112] (or Double Fetch [116]) bugs. Race conditions can further cause critical attacks [113]. However, not all introduced race conditions are security-critical, requiring us to confirm their impact manually.
- Introducing other impacts. There could be other security impacts such as side-channel attacks. To handle those cases, we manually analyze them individually.

2.2 Methodology of Bug Collection

To comprehensively study *CISB*, we first need to collect a substantial set of real compiler-introduced security bugs, as no such dataset exists. Our rationale for collecting *CISB* is that when a compiler-introduced bug is discovered, in general, two things would happen: either the compiler maintainers fix the compiler issue, or the compiler users modify their code to avoid the compiler optimizations. We collect compiler-introduced bugs (mixed with compiler bugs) from the following two sources.

Bugzilla for compilers. First, we collect bug reports that users submitted to the compiler bug trackers (GCC-Bugzilla [36] and LLVM-Bugzilla [62]). In general, we find that they include substantial information about different kinds of potential compiler issues. However, most cases are not in our study scope; compiler users often report non-compiler cases or cases related to compiler correctness. It is worth noting that this project considers cases orthogonal to compiler correctness; interestingly, such cases are usually marked as *INVALID* [98] by compiler developers (since their Bugzilla is

set to track compiler correctness bugs). Among these cases, some are indeed related to *CISB* (our targets) while most of them are not; so we manually review all reports to identify cases that introduce security bugs.

Program patches related to compiler issues. To quickly avoid erroneous compiler optimizations, compiler users often change their source code to avoid the optimizations (instead of trying to change compiler behaviors). In this case, a patch will be issued and typically contains information on how compilers may introduce bugs. Therefore, we use the patch history of popular OSS programs as the second bug source. In particular, we pick the Linux kernel as the target because it has a huge and diverse code base with many corner cases that would trigger optimization issues.

Combining the aforementioned methods, we initially collected 9,334 GCC Bugzilla bug reports, 1,443 LLVM Bugzilla bug reports, and 998,963 unique kernel commit logs. To further refine the bug report set, we propose the following method to find the most relevant *CISB* cases.

- We first select related Bugzilla reports based on their attribute values. Attribute values are assigned when the compiler user submits the bug report to Bugzilla. For example, we observe that, for a *CISB*, when asked to assign which component of the compiler has the bug, compiler users tend to choose values like “-New Bugs” or “LLVM Codegen”.
- We further use the cross-keyword strategy to select the most relevant git logs of the Linux kernel. That is, we define some groups of keywords; each represents similar cases. We can use the intersection of git logs in different groups to discover the most relevant cases. Table 1 shows some of our intersection results.

This scheme allows us to target the most relevant cases. Finally, we select 1,601 cases from Bugzilla and 3,226 from the Linux patch history for manual analysis. This scale is larger than other existing studies that require manual inspection [77, 89, 120], which involves hundreds of StackOverflow pages or GitHub commits.

Table 1: The result of keyword intersection in the Linux kernel (5.16) The grey cells represent the git commits we collected. We use regular expressions such as “[Gg]cclGCC” for keyword “gcc”(The details are shown in Table 8 of Appendix).

	gcc	clang	compiler	optimize	security	attack
gcc	9,658	-	-	-	-	-
clang	-	3,285	-	-	-	-
compiler	-	-	9,101	-	-	-
optimize	651	159	883	11,696	-	-
security	138	62	114	112	6,356	-
attack	21	9	24	31	152	1032

2.3 The Bug Set

From this labor-intensive study, we collect 122 *CISB*; 68 from 1,601 Bugzilla reports, and 54 from 3,226 Linux patches. This sample set is similar to other bug study works involving manual inspection e.g., 146 bugs in [89], 109 bugs in [44]. These bugs can be summarized as 48 unique *CISB* types. (We group bugs with the same cause and impact into a unique bug type.) From them, we reproduce 34 unique *CISB* types. Vulnerability reproduction is laborious [73], and reproducing *CISB* is often harder, as it requires triggering certain compiler behaviors while avoiding the effects of unrelated optimizations. We were unable to reproduce the remaining cases due to the lack of triggering environment settings or because they were related to some architectures that we were unable to test.

Bug Distribution. Table 2 shows the number of security bugs for each class (see the classification in Table 4). First, *CISB* are much more diverse than previous studies [30, 108] thought. Second, we found fewer Orthogonal Specification (defined in §3.1) cases in Bugzilla, probably because users know such situations and choose not to report them to Bugzilla. However, these cases are also common, according to their distribution in the Linux patch history. Table 3 shows the distribution of *CISB* by year, which indicates that the incidence of *CISB* in recent years trends higher than in earlier years. Additionally, the universal application of compilers and optimizations increases the impacts of *CISB*. Lots of software can be affected, such as the Linux kernel, impacting many end-users.

Security Impacts of *CISB* in Real World. The *CISB* we found can lead to different types of security impacts. Specifically, 20.5% can cause system hanging (DoS), 49.2% lead to crashes, and 20.5% lead to information leaks. The remaining 9.8% are related to security check bypassing, which may lead to, e.g., memory errors or information leaks, depending on the guarding checks.

Dataset Sharing. Our dataset is available online, including test cases and their triggering oracles for all the reproduced *CISB*. This dataset can benefit the research community in several ways. In addition to helping researchers to develop and evaluate new techniques for *CISB* detection and prevention, the *CISB* cases can serve as educational and training materials for students and junior analysts.

3 RQ1: General Causes and Classifications

3.1 Fundamental and General Causes

To answer why a *CISB* happens requires answering why the compiler’s transformations dismantle the security property specified by programmers. Based on all the cases we found in the study, we summarize them as two general root causes of *CISB*. In summary, security properties can be represented by source-level abstraction [80], but are orthogonal to the functionality of semantics, while the correctness of programming

languages like C/C++ lies in maintaining semantic equivalence. Therefore, the security boundary in the source is invisible to current compilers.

Root Cause 1: Implicit Specification (ISpec). The compiler makes assumptions that conflict with the functionality of the code in respect to the security property. This allows the compiler to perform aggressive optimizations to dismantle the security property. For example, the compiler can assume Undefined Behavior does not exist, while programmers may rely on the execution result of the Undefined Behavior to ensure the required security property. We refer to the specification of these assumptions as *Implicit Specification (ISpec)* as they implicitly disallow the functionality which the explicit execution result can verify. Such specifications can be those of the language specification or other de-facto ones of the compiler. In Figure 2, the compiler user blames the compiler for eliminating the null pointer check on line 6. The conflict here is that the compiler assumes that a null pointer dereference does not exist while the programmer leaves a null pointer dereference at line 4 as it is useful and does not cause a segmentation fault in some environments such as embedded ARM devices. Based on this assumption, as the pointer *b* is dereferenced at line 5, the compiler infers that the pointer cannot be null at line 6. Such a removed check will propagate the null pointer and may endanger the system. For example, accessing such an escaped null pointer may be exploited to gain execution control by rewriting the interrupt table [10].

```
1 /* GCC bugzilla id:33629 */
2 void bad_code(void *a)
3 {
4     int *b = a;
5     int c = *b;
6     if (b) {
7         ...
8     }
9     ...
10 }
```

Figure 2: A GCC Bugzilla case showing that the compiler assumption of undefined behaviors eliminates necessary security checks.

Root Cause 2: Orthogonal Specification (OSpec). The security property exceeds the semantic functionality scope of language specifications. We refer to the theoretical specification required to preserve security during compilation as an *Orthogonal Specification (OSpec)*; they are orthogonal to those of correctness. Such security properties can be the execution time or the lifetime/region of sensitive data. For example, Figure 1 shows a typical case in the Linux kernel where the compiler introduces an information leak, as the confidentiality of sensitive data is orthogonal to semantic functionality. The `memset` should clear the sensitive buffer after its last use to prohibit a memory disclosure vulnerability from revealing the secret. Unfortunately, the compiler may eliminate it when performing *dead store elimination (DSE)*, because the newly stored value is never used. The sensitive data then persists in memory and may be disclosed by an attacker or captured in a memory dump, thus leaking information. Note that such

Table 2: Statistics of bugs reported to Bugzilla and in the Linux kernel. Table 4 shows the classification. *ISpec* is short for implicit specification and *OSpec* is short for Orthogonal Specification (both defined in §3.1). We attribute bugs with the same cause and impact to a unique kind.

Bug class	bugs			Bugzilla bugs		Kernel bugs	
	Unique bugs	Total	Proportion	Unique bugs	Total	Unique bugs	Total
ISpec1: Eliminating security related code	18	60	0.49	13	50	8	10
ISpec2: Reordering order-sensitive security code	7	10	0.08	1	4	6	6
ISpec3: Introducing Insecure Instructions	17	25	0.20	5	12	12	13
OSpec1: Making sensitive data out of bound	3	21	0.17	2	2	3	19
OSpec2: Breaking timing guarantees	2	5	0.04	0	0	2	5
OSpec3: Introducing insecure micro-architectural side effect	1	1	0.01	0	0	1	1
All	48	122		21	68	32	54

Table 3: Temporal distribution (report date) of bug classes. Bug classes are like in Table 2. The 04-06 refers to the year 2004 to 2006.

Years	04-06	07-09	10-12	13-15	16-18	19-21
ISpec1	1	12	14	10	11	12
ISpec2		2		2		6
ISpec3	3		4	4	10	4
OSpec1	1	1		9	4	6
OSpec2		2		2	1	
OSpec3						1
All	5	17	18	27	26	29

security specifications form a general abstraction of program specifications. They are related to general security properties but not limited to a specific program.

3.2 Three-Layer Classification

While there are two general root causes, in our study we observe that they can lead to various insecure compiler behaviors that result in security consequences. We propose a *three-layer taxonomy*, as shown in Table 4. This classification aims to help understand *CISB* from different perspectives: root causes (§3.1), insecure compiler behaviors, and security consequences. See corresponding examples in §4 and §5.

4 RQ1: Implicit-Specification

This section explains the causes of *CISB* of *Implicit Specifications (ISpec)*: how these bugs happen and their impacts according to our taxonomy and real-world cases.

Causes. As shown in §3.1, the root cause of *ISpec* cases is the conflict between compilers’ implicit assumptions and source code functionalities (*ISpec-conflict*). In particular, based on our identified cases, the assumptions involved in the *ISpec-conflict* can be divided into three classes.

- *No-UB*, is the most common type of *ISpec* where compilers assume that the source code is free of Undefined Behavior (*UB*). Language specifications permit this assumption to reduce compiler complexity and help compilers produce faster code. A conflict arises when programmers rely on the execution of *UB* for the functionality of their code. Note that *CISB* of *No-UB (UB-CISB)* is different from *UB* bugs:

triggering *UB* in the source code of *UB-CISB* will not cause any correctness or security issues.

- *Default-behavior*, is the type of *ISpec* where compilers decide it is appropriate to perform certain default behaviors or make default assumptions. For example, compilers may promote other types to `int` (integer promotion), and compilers assume functions always return by default. A conflict arises when programmers expect that such *Default-behavior* does not happen in their source code.
- *Environment*, is the type of *ISpec* where compilers assume some implementation details of the environment such as whether an instruction type should be used or memory alignment of subtypes. The conflict arises when programmers write tailored code on their target machine which does not match the *Environment*.

How a security bug happens. The *ISpec-conflict* acts only as a prerequisite for a *CISB*. The bug will not happen until (i) the compiler obtains additional information from inferences based on such *ISpec* assumptions, (ii) the compiler performs optimizations based on such information and breaks the functionality of the security property. The aggressive inference based on *ISpec* can be about the range of a variable’s value or the ordering relationship of two operations. Such information can help compilers infer redundant code or candidate instructions for reordering or replacement.

Insecure optimization behaviors. For *ISpec* cases, the security property should be in the functionality scope of compiler specifications. Then insecure optimization behaviors that break the security property must be a modification of functionality, i.e., it can be the elimination, reordering, or introduction of code actions. We will now (i) discuss real-world cases for a concrete look at *ISpec* cases, i.e., showing which inference is used and the concrete *ISpec-conflict*; (ii) demonstrate their security impacts.

4.1 Eliminating Security-related Code

Code elimination is a class of common optimization behaviors such as Dead Code Elimination or Dead Store Elimination; the elimination is performed based on the inference of values, relations, or the status of operations. A *CISB* happens when the inference is derived from *ISpec* and conflicts with the

Table 4: A three-layer taxonomy of compiler introduced security bugs.

Root cause	Insecure optimization behaviors	Security consequences
Implicit Specification §4 (<i>No-UB, Default-behavior,</i> <i>and Environment</i>)	Eliminating security related code §4.1	Elimination of security checks Elimination of critical memory operations
	Reordering order-sensitive security code §4.2	Disorder between order-sensitive memory operations Disorder between security checks and dangerous operations
	Introducing insecure instructions §4.3	Introduction of invalid instructions of certain environments Introduction of insecure logic
Orthogonal Specification §5	Making sensitive data out of bound §5.1	Violation of sensitive data’s living-time boundary Violation of sensitive data’s space(memory) boundary
	Breaking timing guarantees §5.2	Introduction of the time side channel Disordered concurrency sequence due to modification of duration
	Introducing micro-architectural side effects §5.3	Introduction of bounds check bypass vulnerability

source code. For the cases whose security-related code has been eliminated, we further divide them based on the specific security consequences, including eliminating security checks and critical memory operations.

Elimination of security checks. Security checks [64] are conditional statements used to check the value or status of some variables to further ensure security. Common security checks include NULL checks, bound checks or permission checks. When being eliminated, their protection for these security guarantees is broken.

The security impact depends on what the eliminated check protected. One common impact is introducing an infinite loop; it happens when the condition check is used as a loop check. As for those caused by *No-UB*, the most common impact is introducing an unexpected *UB* such as null pointer dereference, division by 0, or out-of-bound access. Next, we discuss real cases by the involved inference and *ISpec-conflict*.

(i) *Value range inference via No-UB.* With such an inference, the compiler can decide a security check is redundant. Typical cases include assuming no signed integer overflow and no null pointer dereference (such as shown in Figure 2).

(ii) *Same value inference via Default-behavior.* The compiler can eliminate subsequent checks of the same value with such an inference. As in a case on the Linux kernel [47], the value can also be the return value of a function. The *ISpec-conflict* exists in this case when the compiler infers that two adjacent callings of function `have_cpuid_p()` share the same return value while the value may be affected by global. That is because the function may access flag registers in the inline assembly code and flag registers are “global”.

Elimination of critical memory operations. The eliminated security code may be critical memory operations. This is mainly caused by implicit inferences against some memory objects and incorrectly determining necessary memory operations as redundant.

The security impacts are mostly memory corruptions, e.g., uninitialized memory access when initialization is eliminated. As the value of the victim variable is modified, sometimes it may break the intended logic and introduce specific insecure logic, like bypassing security checks [39].

(i) *Redundant memory-operation inference via No-UB.* Com-

pilars can make such inferences when the memory operations involve one or more *UB*. For example, compilers assume no modification against a constant variable and no strict alias violation. However, developers may need to modify the constant variable to do the initialization [29]. In this case, this trick works for GCC with a memory barrier but Clang will determine the barrier redundant and remove the initialization. Compilers also assume no data race, based on which compilers can falsely decide unused objects and remove operations such as initialization against them [85].

(ii) *Value not-used inference of Implicit Default-behavior.* A typical *ISpec-conflict* exists when a memory cell is only read in the asm block but not listed in the “Input Operands”. According to the specification [37], if a value is in the “Input Operands”, it is read in the asm block. However, the compiler infers that if the value is not in the “Input Operands”, then it is not read in the asm block. Such an implicit inference is not logically equivalent to the specification. Note if the inference is explicitly expressed in specifications, then the source allows “Value not-used inference” by specifications. This case is therefore not *CISB* but a source bug. One such case in the Linux kernel [12] results in a file system failure.

4.2 Reordering Order-sensitive Security Code

Some security-related operations have a strict ordering requirement; breaking the ordering may cause security issues. This behavior is often caused by the compiler’s improper inference of the relation among some operations based on *No-UB* and *Default-behavior*. The security consequences depend on which part of the code is reordered.

Disorder between order-sensitive memory operations. Due to *No-UB*, compilers may make inferences to judge the relation among memory operations and reorder them. However, such reordering may violate security rules, leading to security bugs. We found that such disorder can easily cause memory corruption, e.g., when the initialization of a variable is reordered to be after its uses, which leads to uninitialized uses. This can also introduce data races and corruption. We found that common related compiler assumptions are no strict-aliasing violation and no data races,

and we found ample security bugs caused by these assumptions [2, 28, 35, 40, 58, 101, 107, 115].

Disorder between security checks and critical operations. Commonly, critical operations are executed only when they pass security checks. Therefore, the reordering of them would immediately result in the bypassing of the security checks. In the case shown in Figure 3, the order of the check in line 2 and the division in line 7 must be ensured. The compiler by default assumes the function `ereport` to return. That is, line 7 will always be executed no matter which branch is taken and `arg2` is unchanged between line 2 and line 7. Based on such inference, the compiler of some architecture will move the division at line 7 ahead of the check at line 2, eventually bypassing the check, introducing a new division-by-zero and terminating the application.

```
1 /* debian bugreport id:616180 */
2 if (arg2 == 0)
3     ereport(ERROR,/* will not return */
4             (errcode(ERRCODE_DIVISION_BY_ZERO),
5              errmsg("division by zero")));
6 /* arg2 cannot be 0 here */
7 result = arg1 / arg2;
```

Figure 3: A case about function return assumption caused security check reordering. GCC on SPARC64 moves the division in line 7 before the check in line 2, resulting in a divide-by-zero issue.

4.3 Introducing Insecure Instructions

Compilers can also introduce insecure code via implicit specifications (e.g., the assumptions of alignment or the layout of structures), thereby undermining security guarantees. We again divide this class based on the security consequences of such translation.

Introduction of invalid instructions of certain environments. Compilers can produce invalid instructions when the actual environment differs from the assumed one or when the default behavior does not fit a real scenario.

Based on such inference, the compiler can produce otherwise invalid instructions in a certain environment. A typical case in our study is that compilers may produce unaligned access on align-required architectures or use align-required instructions to perform unaligned access. As is shown in an LLVM case [25], the compiler uses a `VLDLDR`, an align-required instruction on ARM Cortex-M4, to perform a faster 4 byte read even when the memory access is unaligned. A hardware crash happens when a `VLDLDR` is performed in an unaligned way. *Functionality inference via Default-behavior.* Based on such inference, compilers can produce insecure logic. For example, on GCC Bugzilla [100], the compiler replaces a calling of `printf()` to `puts()` with such inference. However, `puts()` does not allow `NULL` as a parameter while `printf()` does. As a result, the compiler optimization causes the program to crash when `puts()` receives a `NULL` parameter.

Introduction of insecure logic. Compilers can make as-

sumptions that do not fit a real scenario and further modify the explicit user-expected logic in the source code. Such transformation may introduce security problems when the original logic is modified to an insecure one.

Standard function inference via Environment. A *ISpec-conflict* may exist when programmers define their own standard functions. We found such a case in the Linux kernel, as shown in Figure 4. The compiler assumes that `memset()` must return, and the return value is its first argument, i.e., the pointer `waiter`. As the return value of ARM functions is stored in the register `R0`, the compiler can further infer that the value of `waiter` is stored in `R0` after the call of `memset()` in line 5. With these inferences, the compiler replaces `waiter` with `R0` in line 6 and line 7. However, this kernel version of `memset` should not return any value. Therefore, `R0` can be used for other purposes and cannot represent the `waiter` in line 6 and line 7; changing this pointer to `R0` will cause memory corruption.

```
1 /* commit: 455bd4c430b0c0a361f38e8658a0d6cb469942b5 */
2 void debug_mutex_lock_common(struct mutex *lock,
3                              struct mutex_waiter *waiter)
4 {
5     memset(waiter, MUTEX_DEBUG_INIT, sizeof(*waiter));
6     waiter->magic = waiter;
7     INIT_LIST_HEAD(&waiter->list);
8 }
```

Figure 4: A case about the redefinition of standard function causing memory corruption. The compiler assumes the return value of `memset()` at line 5 is in the register `R0` and uses it to replace the `waiter` in line 6,7; this will cause memory corruption.

Default-behavior assumption. Security-related logic can also be modified due to *Default-behavior*. A typical scenario is that security checks are invalidated due to automatically promoting types. We found multiple cases in which compilers automatically promote data types like `char` and `short int`, which take fewer bytes than `int`, to `int` or `unsigned int` when an operation is performed on them [87]. Such “int promotion” is the default compiler behavior permitted in the language specification. In this case, `if (u8 + 1)` is used to catch the situation when the unsigned 8-bit number `u8` is 255 by legal unsigned 8-bit overflow [95]. However, when “Int Promotion” happens, the check fails because `(255+1)` is a normal integer number, and there is no overflow to make `(u8 + 1)` be 0. Such a failed check can be dangerous, e.g., when it is used as a loop check. If it happens, an infinite loop will break availability.

No-concurrency assumption via No-UB and Environment. C/C++ compilers can do aggressive optimizations that are correct for single-threaded execution, but dangerous in a concurrent environment due to the lack of explicit specification of concurrency. With C11, C specifications include a detailed memory model to better support concurrency, defining concurrency-related *UB*. Programmers must use new primitives to handle concurrency in their code, or their code will be prone to *UB*. Suddenly, legacy code can be blamed for

containing *UB*. Different from other *No-UB* but similar to Orthogonal-Specification cases, here compilers assume a no-concurrency environment but not specific kinds of *UB*. An *ISpec-conflict* exists when programmers arrange their threads well but without provided primitives. For example: (a) a deep copy can be optimized into a shallow copy for performance reasons, breaking consistency. As Figure 5 shows, the compiler eliminates the deep copy of *vma*, leaving a time window between line 5 and the use of *vma*. Because the later use of *vma* has been replaced with the origin memory read of the racy *mm->mmap_cache* (other than of *vma*), the value of *mm->mmap_cache* may have been changed after it has passed the check in line 5, leading to a typical TOCTTOU bug. (b) The atomicity of critical operations can be broken. Compilers may split one critical operation, such as one shared memory access, into several ones, e.g., by splitting a larger object into small ones and then accessing each part separately, the critical operation becomes non-atomic. Such violation of operation atomicity incurs interrupts against concurrent operations, and causes memory corruption or other insecure logic errors. For example, Linux kernel developers found GCC uses `rep movsl` rather than `rep movsq` for a structure copy, breaking the atomicity of the access to such a shared structure; triggering memory corruption [121].

```

1 /* commit: b6a9b7f6b1f21735a7456d534dc0e68e61359d2c */
2 - vma = mm->mmap_cache;
3 + vma = ACCESS_ONCE(mm->mmap_cache);
4   if (!(vma && vma->vm_end > addr && vma->vm_start <= addr)) {
5     struct rb_node *rb_node;

```

Figure 5: A double fetch bug introduced by GCC 4.8 on s390x. `ACCESS_ONCE()` is used to force the compiler to do the deep copy.

5 RQ1: Orthogonal-Specification

For *ISpec* cases, the corresponding specifications are present but implicitly break security properties. Contrary to *ISpec*, our study reveals that security-related specifications can be completely omitted, as they are often orthogonal to correctness-related specifications that compilers emphasize. According to our study, Orthogonal Specification (*OSpec*, as defined in §3.1) can also cause critical security impacts. We divide *OSpec* cases by the insecure optimization behaviors and security consequences they bring.

5.1 Moving Sensitive Data Out of Boundary

We define a boundary violation of sensitive data as the existence of sensitive data in memory exceeding the boundaries of space/time the developer intends to enforce. Due to security boundaries for sensitive data not included in specifications, compiler optimizations may break the intended boundaries and lead to leaks or corruption of sensitive data.

Violation of living-time boundary of sensitive data. This problem occurs when the sensitive data persists in memory

longer than its originally designed lifetime because of compiler optimization. Most sensitive data, such as secret keys, urge such enforcement of the boundary. Common security impacts include leaks or corruption of sensitive data due to the expansion of attack surfaces.

One representative case is the elimination of secret scrubbing during the Dead Store Elimination (DSE) optimization. Information leaks occur when compilers try to scrub some sensitive data such as passwords or cryptographic keys. Previous work [9, 103, 119] also discussed this problem in detail.

Violation of sensitive data’s space (memory) boundary.

This problem occurs when the sensitive data reaches out of the memory region its developer intends to enforce because of compiler optimizations. Such enforcement of the memory boundary of sensitive data is common in systems with multiple memory layers of different privileges such as the user space and kernel space in the OS kernel. Similarly, they can cause leaks or corruption of sensitive data. In general, passing uninitialized memory within a single address space does not violate security; however, it becomes serious information leaks when data passes past the kernel/user boundary. The common cases are uninitialized structure padding and partial union initialization introduced by compilers. Passing the uninitialized memory introduces information leaks, as shown in [65]. Unfortunately, this problem remains severe with several recently observed cases in the Linux kernel [5, 8, 14].

5.2 Breaking Timing Guarantees

Compilers are oblivious to timing guarantees (e.g., the same execution time of two paths) enforced by developers, and optimizations may destroy guarantees. For cases in this category, we divide them based on the specific security consequences.

Introduction of the timing side channel. Compilers may break the timing requirement and introduce timing side channels by many optimizations [30], e.g., the secret can be inferred from the time information of the crypto code. For example, the compiler makes the (Linux version) `memcmp()` return early when a mismatch of the memory comparison occurs, which may leak timing in crypto code [43]. In this case, the developers disable this optimization on `memcmp()` using the compiler option `-Os`. However, this patch is revised later [6], as disabling compiler optimizations is fragile. As adding new optimizations to `-O0` or `-Os` would break the assumptions the code is making.

Disordered concurrency sequence due to modification of duration. A time delay can be used to ensure the concurrency sequence, such as waiting a given duration for some hardware initialization progress to complete and then using the hardware; however, compiler optimizations can change the duration, and then the sequence is disordered. A typical security consequence of such a disorder is bypassing a security check, such as a case [70] in the Linux kernel.

5.3 Introducing Insecure Micro-architectural Side Effects

Typically, a speculative execution side channel can be used to leak sensitive information [46, 59]. Code sequences with certain features are vulnerable to this side-channel attack. Due to the unawareness of such side effects, compilers may add such vulnerable features and bring side channel attack surface to created code.

Introduction of bound-check bypass vulnerability. Several code sequences can be used as the gadget to craft speculative execution side-channel attacks; bound-check bypass vulnerability is one of them [42]. It can cause the leakage of sensitive data through cache-based side-channel attacks. Compilers can introduce speculative execution in bound-checks, as shown in Figure 6 and this blog [41]. The `switch` can be optimized to lines 9 and 10. Line 10 can now be speculatively executed regardless of the check at line 9, which can finally leak arbitrary information in memory through cache-based side channel attacks as the method in the Meltdown paper [59]. The root problem is that the optimization introduces micro-architectural side effects, and compilers are unaware of it.

```

1 /* Before optimization*/
2 switch(x) {
3     case 0: return y;
4     case 1: return z;
5     ...
6     default: return -1;
7 }
8 /* After optimization*/
9 if (x < 0 || x > 2) return -1;
10 goto case[x];

```

Figure 6: An example of compiler-introduced speculative bound check pass caused information leaks. A Bound Check Bypass vulnerability is introduced by the compiler’s processing of `switch`.

6 RQ2: User Study and Survey

In general, compiler users are expected to prevent *CISB*. Given a large number of compiler rules and assumptions, is it realistic for users to correctly follow them and avoid *CISB*? We conduct a user study to answer this question from three angles: A1: the programmers’ knowledge and awareness of *CISB* and related issues, A2: the programmers’ first-hand experience or estimates of experience of encountering a *CISB*, and A3: programmer views of *CISB*. In addition, we also ask for their expectations for prospective research.

Procedure of the user study. We select the most common *UB-CISB* and *CISB* of *OSpec* for the study. (*UB-CISB* means *CISB* caused by *No-UB* assumption, and *CISB* of *OSpec* means the *CISB* caused by *OSpec* (as defined in §3.1).) To ensure that the participants precisely understand the concepts in our study, we provide reading materials before questions. Our reading materials include learner-friendly definitions of *UB*, *CISB*, *UB-CISB*, *CISB* of *OSpec*, and three typical *CISB* cases.

Table 5: Table of Participant Demographics

		Survey (n = 62)	
Organization	Compiler Developer Communities	9.68%	
	Company1	17.74%	
	Company2	11.29%	
	Company3	3.23%	
	Company4	1.61%	
	Unknown Company	1.61%	
	University1	22.58%	
	University2	17.74%	
	University3	11.29%	
	University4	1.61%	
	Unknown University	1.61%	
	Role	security analyst / security maintainer	6.45%
		compiler developer or maintainer	3.23%
professional C programmer		30.65%	
academic researcher		22.58%	
master student		11.29%	
PhD student		22.58%	
OS developer		1.61%	
Others		1.61%	

We also provide a *UB* bug to help programmers learn the difference between *UB-CISB* and *UB* bugs. After the knowledge questions, we also briefly introduce mitigations and ask for their opinions on who should bear the responsibility of the two kinds of *CISB*. We further ask for their estimates of the difficulty to learn, debug and avoid writing *UB-CISB*. Finally, they are asked to express their expectations for prospective research in this area.

Recruitment. After obtaining an ethics review waiver from the local ethics review board ², we recruit participants by distributing recruitment advertisements online to 4 universities and 4 companies (see the detailed requirements for recruitment in Appendix 12.1), contacting compiler developers or maintainers, and snowball sampling where participants recommended other colleagues. In total, we recruit 62 participants, including 27 industry employees, 14 academic researchers, and 21 graduate students with a background in system security or programming languages. Table 5 details the demographics of participants. Participants had an average of 7 years of C programming experience. We follow a standard and ethical way to reward participants (with gift cards), and 6 volunteers declined the rewards. We believe the number of participants is substantial, as it is already more than 12-20 participants as suggested by qualitative research best practices literature [38] and also aligns with related works [73, 104].

Results and findings. Here we present the main findings. *A1: Programmers’ knowledge and awareness:* (1) There are knowledge gaps among programmers’ knowledge of *UB*, *No-UB* assumption, and *UB-CISB*. As shown in Figure 7, the proportions of their knowledge are 90.3%, 48.4%, 41.9%, respectively. (2) C programmers do not know *CISB* well, especially for *UB-CISB*. Only 41.9% of them know about

²This study only includes interactions involving survey procedures. No information which could identify the human subjects will be shared or stored with the data.

UB-CISB, and 50% know about *CISB* of *OSpec*. (3) C programmers do not know the *UB-CISB* mitigations provided by compilers well, 41.9% of them do not know such mitigations and 48.6% have heard about them but do not know the details. *A2: Programmer experience of UB-CISB*: (4) It usually takes a lot of time to solve *UB-CISB*. As shown in Figure 8, over half of the programmers either took more than 2 hours to solve a *CISB* or even did not manage to solve it. (5) C programmers often consider it difficult for them to learn and understand, debug or avoid a *UB-CISB*, as shown in Figure 9; Interestingly, from some feedback, we found that some programmers assume and trust the security of compilers. A hard part for them in the first place is realizing that these problems are silently introduced in binary code by the compiler.

A3: Programmer Views: (6) C programmers tend to agree that the compiler should take a bit more responsibility for most *CISB*. When asked to rank the degree of responsibility from 1 to 9 (1 means completely the responsibility of the programmer, 5 means neutral, 9 means completely the responsibility of the compiler), they rank *UB-CISB* 5.81 and *CISB* of *OSpec* 5.87 on average. (7) Most C programmers think it is necessary to collect all kinds of *CISB* behavior systematically. 82.3% of them in our survey take it as “Extremely necessary” or “Very necessary”.

Summary: The quantitative results of the above three aspects show that it is unreliable to expect programmers to prevent *CISB* themselves in the current situation.

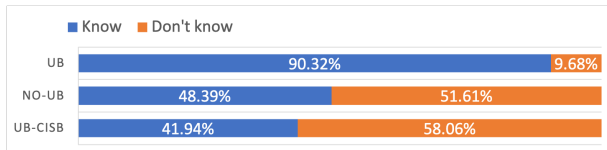


Figure 7: The percentage of C programmers’ knowledge of UB (Undefined Behavior), *No-UB* assumption and *UB-CISB*.

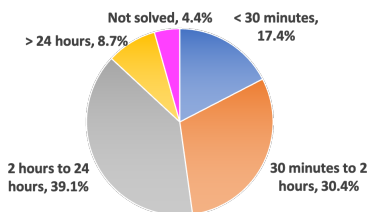


Figure 8: The percentage of the longest time to solve a *UB-CISB*, for C programmers who have encountered a *UB-CISB* (N=23).

Feedback. We received positive feedback from participants, e.g., the study helped them explain some unsolved issues; they encountered a real *UB-CISB* just a few days after participation, and our survey helped them find the root cause.

Bias prevention. We try to avoid biases by: (i) recruiting participants from the compiler communities and inviting compiler developers or maintainers; (ii) making it clear in the survey that our terminology does not represent any subjective

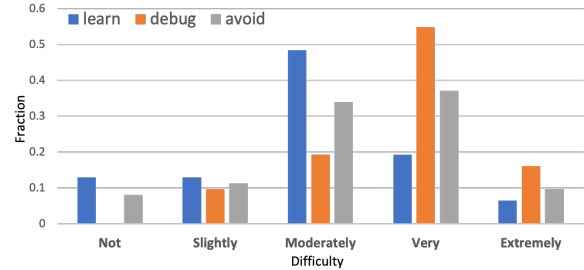


Figure 9: The degree of difficulty in the view of C programmers to (i) learn and understand *UB-CISB*, (ii) debug the root cause by themselves, (iii) avoid writing *UB-CISB* with *No-UB* told and the complete list of *UB* rules.

assessment and encouraging them to share their own opinions.

7 RQ3: Current Mitigations

Here we study current *CISB* mitigations and highlight any remaining risks.

7.1 Programmer/User Efforts

Typically compiler users are expected to avoid *CISB* themselves. Compiler users usually have three choices: (1) Normalizing the source code to avoid potential victim semantics, such as undefined behaviors. (2) Using language level or hardware level mechanisms such as *volatile* or memory barriers to control (force or block) compiler behaviors. It is ad-hoc and unstable to tame compiler behaviors, but sometimes users have limited choices. System-level programs or libraries often provide some primitives as interfaces based on these mechanisms. This is to provide unified and specialized solutions for programmers and relies on experienced system developers to handle complex compiler behaviors. Note that such primitives may still fail, as shown in many Linux kernel cases (e.g., Figure 1). (3) Blocking all compiler optimizations. As the strategy *O0* in Table 6, blocking optimizations is not always effective (< 95%), and the performance overhead is prohibitively high (> 2.5X). Our study reveals that it is risky to rely on users to avoid *CISB* for the following reasons (in addition to the quantitative evidence from our user study).

UB rules are hard to learn and understand. There are 200+ rules in the C standard (ISO C17 J2) for *UB*. Other than those simple programming errors, such as buffer overflow, use after free, about 180 may involve *UB-CISB* [68]. It is impractical for compiler users to remember and follow all these rules. Even given the full list of *UB*, these rules can be easily misunderstood or ignored.

Compiler optimizations may further disable prevention. Unexpectedly, we found that compiler optimizations may silently break the prevention of a *CISB*. Based on our study, we have found many such cases where a *CISB* cannot be fully

patched at one time. As we have discussed in Figure 1, this patch failed when GCC performed a new optimization [66], and failed again when the compiler changed to LLVM [26].

7.2 Compiler Assistance

While generally users are expected to avoid *CISB*, compilers provide some compilation options for preventing bugs or reporting warnings. However, based on our study, we found that such mitigations do not work well: suffering from effectiveness and performance issues.

To understand the issues, we set up an experiment. We take all the reproduced cases in our dataset as representative examples for real-world *CISB*. We select different compiler mitigation strategies and study their effectiveness and performance, as listed in Table 6. The attack target is to ensure that the *CISB* occurs and is not caught by mitigations. The results confirm that (1) The current mitigations are insufficient (e.g., “UBSan” and “Wall” misses about 70% of cases) (2) There is a trade-off between effectiveness and performance. Effective mitigations often hurt performance a lot.

It is worth noting that the effectiveness rates of “O3” and “O2” come mostly from compiler warnings.

Table 6: An evaluation of the mitigations provided by the compiler. For strategies, O0-O3: corresponding compiler option; “All-ub”: a group of compiler flags to mitigate all *UB-CISB*. ‘All-cisb’: a minimal group of options (restricting or disabling optimizations) to prevent all the preventable *CISB*; “UBSan”: Undefined Behavior Sanitizer; “Wall”: reporting warnings with flag “-Wall”. “Eff.(UB-CISB)”, “Eff. (all CISB)” means the effectiveness of the strategy on all the *UB-CISB* or *CISB* in our dataset related to the given compiler. We take a strategy as effective if it avoids the bug from happening, or catches the bug at runtime or compilation time. “Overhead” means the performance overhead tested on a benchmark (SPEC CPU 2006), of compiler GCC 12.0.1 and Clang 14.0.0, taking “O3” as the baseline. (See the used compiler options of all strategies in Table 9.)

Strategy		Eff.(UB-CISB)	Eff. (all CISB)	Overhead
O3	gcc	10.0%	9.7%	0%
	clang	26.7%	16.0%	0%
O2	gcc	10.0%	6.5%	5.4%
	clang	26.7%	16.0%	2.1%
O1	gcc	30.0%	25.8%	21.4%
	clang	26.7%	16.0%	3.7%
O0	gcc	78.9%	67.7%	211.4%
	clang	93.3%	80.0%	170.0%
All-ub	gcc	53.3%	-	17.7%
	clang	55.0%	-	7.2%
All-cisb	gcc	75.0%	61.3%	23.8%
	clang	53.3%	48.0%	8.1%
UBSan	gcc	30.0%	-	186.8%
	clang	33.3%	-	227.7%
Wall	gcc	20.0%	-	-
	clang	26.7%	-	-

7.3 Automatic Prevention

Compared with educating developers to follow all the rules to avoid *CISB*, a more attractive choice is automatically handling them. There are mainly three kinds of automatic *CISB* prevention. We investigate them with our taxonomy; see Table 7.



Runtime sanitizer. Sanitizers add checks during compilation to catch security problems at runtime. UBSanitizer [21] detects common undefined behaviors such as signed integer overflow, pointer overflow, or missing return statements. Some sanitizers are designed to detect other undefined behaviors: ThreadSanitizer [92] can detect data races; TypeSanitizer [32] can detect strict aliasing violations. In addition, Song et al. [96] conduct a survey of sanitizers for security. Fuzzing tools often leverage runtime sanitizers to detect faults [11, 18, 33, 34, 48, 84]. Sanitizers, in general, should not have FP, but FN are unavoidable. To find a bug, it must be executed (i.e., covered) and the sanitizer instrumentation must detect it. The overhead is the leading factor limiting their widespread use.































Formal secure compilation. This class of fundamental solutions enables compilers to preserve security properties during optimizations. Some works develop compilers that preserve given security properties, such as confidentiality [9] and integrity of values, constant execution time [3, 7, 15, 93, 111], or micro-architecture side-effects [82]. Some works provide a formal verifier to guarantee confidentiality [94] or constant-time security [4]. In addition, fully abstract compilation [1, 81, 105, 106] aims at ensuring observational equivalence during compilation and thus is a general solution for secure compilation. Patrignani et al. [80] survey that. The performance and complicated use are factors limiting their widespread use. However, the efficiency of securely compiled code is rarely considered [80].

Dedicated security analysis. Many works provide a specific abstract model or focus on semantic patterns to identify certain security problems. A few of them are specially designed for *CISB*: STACK [109, 110] detects *CISB* in the scope of *ISpec* 1 (§4.1) related to *UB*; Yang et al. [119] and Sprundel [103] identify *CISB* of *OSpec* 1 (§5.1) related to the elimination of secret scrubbing by semantic patterns. Most of them focus on broader security problems such as cryptographic key misuse [55], information leak bugs [65], Undefined Behavior bugs [60, 114], Spectre gadgets [79, 86]. Certain kinds of *CISB* are included but not the direct target, e.g., Unisan [65] can prevent information leak bugs including *CISB* of uninitialized structure padding (§5.2). All such detection works suffer from FN and FP issues.

8 RQ4: Challenges for Future Research

User study. The ability of a developer to avoid *CISB* situations (especially *UB-CISB*) when writing code should be

Table 7: Automatic Prevention works. For types, "SD": "Static Detection"; "DD": "Dynamic Detection"; "RP": "Runtime Prevention"; "C": "New Compiler"; "V": "Verifier"; "F": "Formal Foundation". For Target CIBS, bug classes are like in Table 2, circles like  (100%) mean the maximum percentage of CIBS in our dataset that can theoretically be prevented. For FP&FN, ✓: YES, ✗: NO,  (100%): the known ratio. For PO (performance overhead), ○: < 1%; ◐: < 20%; ◑: < 100% ◒: < 500% ; ◓: >= 500%. Works with ★ means it is specific to CIBS.

Automatic Prevention	Year	Type	Target CIBS	Analysis			
				FP	FN	PO	
Runtime sanitizer	UBSan [21]	2012-	DD&RP ISpec 1,2,3		✗	✓	
	ThreadSan [92]	2009-	DD&RP ISpec 1,2,3		✗	✓	
	TySan [32]	2017-	DD&RP ISpec 1,2,3		✗	✓	n/a
Formal Secure Compilation	Ct-verif [4]	2016	V OSpec 2		-	-	-
	Jasmin [3]	2017	C&V OSpec 2		-	-	
	FaCT [15]	2017	C OSpec 2		-	-	n/a
	Besson et al. [9]	2018	C OSpec 1		-	-	n/a
	CT-wasm [111]	2019	C&V OSpec 2		-	-	
	Simon et al. [93]	2019	C OSpec 2		-	-	
	Barthe et al. [7]	2020	C&V OSpec 2		-	-	
	Patrignani et al. [82]	2021	F OSpec 3		-	-	-
Dedicated security analysis	★ STACK [109]	2013	SD ISpec 1		✓	✓	-
	Unisan [65]	2016	SD&RP OSpec 1		✓	-	
	★ Yang et al. [119]	2017	RP OSpec 1		-	-	
	K-Hunt [55]	2018	DD OSpec 2		✓	✓	-
	★ Sprundel [103]	2018	SD OSpec 1		✓	✓	-
	Wu et al. [114]	2020	SD ISpec 1,2,3		✓	✓	-
	SpecFuzz [79]	2020	DD OSpec 3		✓	✓	-
	SpecTaint [86]	2021	DD OSpec 3		✓	✓	-
KUBO [60]	2021	SD ISpec 1,2,3				-	

measured quantitatively. However, our user study (§6) is not enough to give a precise quantitative estimate of it. Such a statistic would guide future work in secure compilation and programmer education. We therefore call for a user study to measure this. The challenge lies in providing an objective, comprehensive, and representative criterion to evaluate programmers.

Automatic Prevention. (1) *Dedicated security analysis.* The key challenge is that there are not many works specially designed for CIBS and many CIBS remain unsolved. The hardest part of identifying a CIBS by program analysis is giving a suitable oracle to help discern if the bug is security-related and compiler-introduced. Another challenge is that source-level analysis should be optimization-aware to help the prevention, due to the nature of CIBS. In addition, for detection works, both False Positives and False Negatives are hard to avoid. One way to reduce the effects of False Positive is to automatically fix the reported bug sites through instrumentation like Unisan [65]. Then the performance should be considered. Another possible direction is to provide warnings for all potential security problems and filter potential CIBS via the behavior of related optimizations or their exploitability such as whether

the potential bug can be triggered through user-controlled data. (2) *Runtime sanitization.* More tailored sanitizers are needed for other CIBS (especially those unrelated to UB) and code coverage is the main challenge. The challenge for a new sanitizer for security is to ensure that the False Positive ratio is negligible (and ideally zero). (3) *Formal Secure Compilation.* Although existing secure compilation approaches show promising results for particular security models, the application of secure compilation techniques to mainstream programming languages has not yet been achieved. In addition, as shown in Table 7, formal works for UB-CIBS are rare. The challenge is UB is piecemeal and hard to model, which is one of the reasons why UB is introduced to ease compilers. We call for work focusing directly on security boundaries that is compatible with commonly used languages like C/C++.

9 Discussion and Limitations

9.1 Related Security Issues

Compilers can also indirectly cause other security issues.

Simplification of Attacks. Compilers may make the code risk-prone and easier to be attacked when breaking defensive properties. For example, not inlining a function in SMAP-disabled regions of OS kernels may introduce security holes [45]; omitting function frame pointer can help attackers to bypass a check [23]. Compilers can also introduce extra ROP gadgets [13] and facilitate code reuse attacks.

Influence on Security Mechanisms. As security mechanisms are unaware of and orthogonal to compiler optimization, they may become ineffective [57]. For example, the security of binary level CFI implementations [75, 83, 102] is weakened if function signature recovery is incomplete [117]. Moreover, security checks of code reuse defense such as CFI can be bypassed via TOCTTOU attacks when compilers introduce double fetch of those checked values through optimizations such as register spilling [22]. Stack protections like LLVM SafeStack [61] can also be rendered ineffective due to register spilling [16].

9.2 Limitations

False negatives. The results still have false negatives due to the imprecise keyword-based searching and manual analysis. To reduce this threat, three authors analyzed potential bugs separately and discussed inconsistent issues until an agreement was reached. In addition, our data sources cannot cover all CIBS; OS kernels can not represent all code bases. Finally, as an empirical study, we will certainly miss some unknown cases, which is expected because compilers evolve, and there will always be new threats in the future. However, we believe that our taxonomy remains helpful when programmers encounter new CIBS cases.

False positives. In this project, we first tried to confirm each case we met by reproducing the compiler behaviors across various versions of GCC/Clang. Moreover, for the complex cases in the Linux kernel that we cannot confirm, our results are based on the evidence provided by compiler users, such as binary code snippets and running traces. Therefore, we believe the false-positive rate of our result should be meager. But there may still be some false positives when programmers give us incorrect hints of the bug’s existence or security impacts.

Other threats to validity. Our dataset may be biased. First, the data source can not represent all styles of code. In addition to the Bugzilla pages, we investigate bugs from Linux kernel commits. The Linux kernel is a huge program with rich code diversity. However, it is just a representative codebase for operating systems. Second, the manual efforts required to analyze the bugs were large. Unlike other bugs, due to the lack of an accepted definition and the piecemeal nature of *CISB*, *CISB* cases can only be filtered manually from a large number of potential reports. We have spent 1,500 person-hours to get the dataset. In addition, our dataset only targets the C programming language. This is also the choice of many related works [30, 65, 108] in this field. *CISB* in C are fundamental and they are shared with C++. Note that *CISB* can also exist in other programming languages.

10 Related Work

CISB Studies. D’Silva et al. [30] defined the concept of *correctness-security gap* and studied compiler-introduced security challenges qualitatively. They classified the bugs into three classes: information leaks through persistent state, elimination of security-relevant code due to undefined behavior, and introduction of side channels. Our study shows that these three classes collected in their research are incomplete in representing the *correctness-security gap* (theoretically covering 69% bugs in our dataset), and each class of their bugs is illustrated with limited cases. As far as we know, it is the only previous work that studies *CISB* broadly. Wang et al. [108] collected a subset of *CISB* related to undefined behaviors and developed a static checker called STACK [109, 110] to detect bugs caused by the elimination of security-relevant code due to undefined behaviors. While STACK finds some bugs, not all *CISB* types are covered, leaving some classes undetected. Wu et al. [114] listed several optimizations that rely on the *No-UB* assumption in LLVM, hooking LLVM code to detect potential *UB* during compilation time. Such a list of optimizations is obtained from expert knowledge, focusing on a few *CISB* types. Some works also talk about a certain kind of *CISB* of *OSpec*. Lu et al. [65], Yang et al. [119] and Sprundel [103] separately discuss a certain type of *CISB* of sensitive data leakage and survey the existing mitigation. Simon et al. [93] study some *CISB* of side channel risks. Table 7 shows the target *CISB* of these works. In comparison, our quantitative

study is designed to draw a comprehensive picture of *CISB*. We collected different *CISB* types from real bug reports rather than expert knowledge, and also qualitatively investigate the knowledge and views of programmers.

Detection of CISB. As discussed in §7.3, these works [60, 65, 103, 109, 114] aim to detect certain kinds of *CISB*, while our study aims to collect various *CISB* types. Note that we do not pursue more *CISB* but more *CISB* types.

Formal Secure Compilation. As discussed in §7.3, these works [1, 3, 4, 7, 9, 15, 81, 82, 105, 106, 111] are mitigations designed to enable compilers to formally preserve security properties. In comparison, our study focuses on concrete security violations and their mitigations.

Works on compiler correctness bugs. There have been many studies [89, 98] or testing tools [19, 20, 24, 49, 50, 56, 76, 88, 97, 118] on compiler correctness bugs. More compiler testing works can be found in the survey [17] of Chen et al.. Lee et al. [51] study some conflicting compiler assumptions regarding *UB* in LLVM IR and provide a solution to ensure the compiler generates correct code. In addition, many works [27, 31, 52–54, 63, 67] formally verify the functionality or correctness of compilers. However, unlike these works focusing on broken compiler correctness, our study investigates security issues where compilers act correctly (but in ways unexpected to the developer).

Empirical Studies on Software Defects. Much research effort has been made to study fault-related characteristics of software systems other than compilers, e.g., Tan et al. [99] inspect bug root causes, impacts and components to find characteristics within open-source projects, Zhang et al. [120] study deep learning application bugs. In comparison, we focus on software bugs unexpectedly introduced by compilers.

11 Conclusion

“Correct” compiler optimizations may introduce security bugs. Here, we study the *compiler-introduced security bugs*. We comprehensively collect a diverse set of bugs from two sources, with 122 bugs and 68 unique bug types. We propose a three-level classification for the taxonomy of the bugs and perform an empirical study against them, including their causes, impacts, and mitigation. Through this study, we summarize many important findings and lessons learned. For example, the “no-undefined-behavior” assumption has become a nightmare for compiler users, and compiler optimizations may further disable patches for these compiler-introduced bugs. We also perform a user study to understand the awareness, knowledge, and views of users on the compiler rules and assumptions. We found that users often lack knowledge of such security bugs and consider these bugs to be hard for them to understand, debug, and avoid. Most of our survey participants tend to agree that compilers should take some responsibility and call for research work. Instead of relying on compiler users to

fully understand compiler behaviors and properly use them, a more effective solution is to enforce advanced and automated techniques on the compiler side. We hope the study motivates future research on secure compilation.

Acknowledgment

We thank the anonymous reviewers for their feedback. This work was supported, in part, by grants from the Chinese National Key R&D Program (2022YFF0604503), the Chinese National Natural Science Foundation (62032010,62172201), China Scholarship Council, Postgraduate Research&Practice Innovation Program of Jiangsu Province, NSF awards CNS-1931208, CNS-2045478, CNS-2106771, and CNS-2154989, SNSF PCEGP2_186974, DARPA HR001119S0089-AMP-FP-034, AFRL FA8655-20-1-7048, and ERC StG 850868. 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] Pieter Agten, Raoul Strackx, Bart Jacobs, and Frank Piessens. 2012. Secure compilation to modern processors. In *2012 IEEE 25th Computer Security Foundations Symposium*. IEEE, 171–185.
- [2] Jade Alglave, Will Deacon, Boqun Feng, David Howells, Daniel Lustig, Luc Maranget, Paul E. McKenney, Andrea Parri, Nicholas Piggin, Alan Stern, Akira Yokosawa, and Peter Zijlstra. 2020. Who’s afraid of a big bad optimizing compiler? <https://lwn.net/Articles/793253/>.
- [3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-assurance and high-speed cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1807–1823.
- [4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying {Constant-Time} Implementations. In *25th USENIX Security Symposium (USENIX Security 16)*. 53–70.
- [5] Liran Alon and Paolo Bonzini. 2018. KVM: x86: Fix kernel info-leak in KVM_HC_CLOCK_PAIRING hypercall. <https://github.com/torvalds/linux/commit/bcbfbd8ec21096027f1ee13ce6c185e8175166f6>.
- [6] Cesar Eduardo Barros and Herbert Xu. 2013. crypto: more robust crypto_memneq. <https://github.com/torvalds/linux/commit/fe8c8a126806fea4465c43d62a1f9d273a572bf5>.
- [7] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal verification of a constant-time preserving C compiler. *Proceedings of the ACM on Programming Languages* 4, POPL (2020), 1–30.
- [8] Arnd Bergmann and Dmitry Torokhov. 2019. Input: input_event - fix struct padding on sparc64. <https://github.com/torvalds/linux/commit/f729a1b0f8df7091cea3729fc0e414f5326e1163>.
- [9] Frédéric Besson, Alexandre Dang, and Thomas Jensen. 2018. Securing compilation against memory probing. In *Proceedings of the 13th Workshop on Programming Languages and Analysis for Security*. 29–40.
- [10] Federico Biancuzzi. 2007. Embedded problems: exploiting NULL pointer dereferences. https://www.theregister.com/2007/06/13/null_exploit_interview/.
- [11] Tim Blazytko, Matt Bishop, Cornelius Aschermann, Justin Cappos, Moritz Schlögel, Nadia Korshun, Ali Abbasi, Marco Schweighauser, Sebastian Schinzel, Sergej Schumilo, et al. 2019. {GRIMOIRE}: Synthesizing structure while fuzzing. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1985–2002.
- [12] David S. Miller Bob Breuer. 2007. [SPARC32]: Fix over-optimization by GCC near ip_fast_csum. <https://github.com/torvalds/linux/commit/51bcf092917bfaa88d762879d0bbfe7619e8c16c>.
- [13] Michael D Brown, Matthew Pruett, Robert Bigelow, Girish Mururu, and Santosh Pande. 2021. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.
- [14] Dan Carpenter and Bartłomiej Zolnierkiewicz. 2020. fbdev: potential information leak in do_fb_ioctl(). <https://github.com/torvalds/linux/commit/d3d19d6fc5736a798b118971935ce274f7deaa82>.
- [15] Sunjay Cauligi, Gary Soeller, Fraser Brown, Brian Johannsmeyer, Yunlu Huang, Ranjit Jhala, and Deian Stefan. 2017. Fact: A flexible, constant-time programming language. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 69–76.
- [16] CERT Coordination Center. 2020. LLVMs Arm stack protection feature can be rendered ineffective. <https://kb.cert.org/vuls/id/129209/>.
- [17] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A survey of compiler testing. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.
- [18] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.
- [19] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. 2013. Taming compiler fuzzers. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 197–208.
- [20] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed differential testing of JVM implementations. In *proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 85–99.
- [21] Clang. 2020. UndefinedBehaviorSanitizer. <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>.
- [22] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. 2015. 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*. 952–963.
- [23] Kees Cook. 2019. x86/asm: Pin sensitive CR4 bits. <https://github.com/torvalds/linux/commit/873d50d58f67ef15d2777b5e7f7a5268bb1fbae2>.
- [24] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. 2012. Testing static analyzers with randomly generated programs. In *NASA Formal Methods Symposium*. Springer, 120–125.
- [25] Tim Northover Dan W. 2017. Hardfault when dereferencing unaligned float. https://bugs.llvm.org/show_bug.cgi?id=34143.
- [26] Herbert Xu Daniel Borkmann. 2015. lib: make memzero_explicit more robust against dead store elimination. <https://github.com/torvalds/linux/commit/7829fb09a2b4268b30dd9bc782fa5ebec278b137>.
- [27] Maulik A Dave. 2003. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes* 28, 6 (2003), 2–2.

- [28] Maya Erez Dedy Lansky and Kalle Valo. 2019. wil6210: make sure DR bit is read before rest of the status message. <https://github.com/torvalds/linux/commit/f4519fd9375d310c608f9a47e4f3a0579bc94998>.
- [29] Nick Desaulniers and Paul Burton. 2019. mips: avoid explicit UB in assignment of mips_io_port_base. <https://github.com/torvalds/linux/commit/12051b318bc3ce5b42d6d786191008284b067d83>.
- [30] Vijay D’Silva, Mathias Payer, and Dawn Song. 2015. The Correctness-Security Gap in Compiler Optimization. In *2015 IEEE Security and Privacy Workshops*. IEEE. <https://doi.org/10.1109/spw.2015.33>
- [31] Grigory Fedyukovich, Arie Gurfinkel, and Natasha Sharygina. 2014. Incremental verification of compiler optimizations. In *NASA Formal Methods Symposium*. Springer, 300–306.
- [32] Hal Finkel. 2017. The Type Sanitizer: Free yourself from -fno-strict-aliasing.
- [33] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. 2020. {GREYONE}: Data Flow Sensitive Fuzzing. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 2577–2594.
- [34] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 679–696.
- [35] GCC. 2002. GCC Bug 90267. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=90267.
- [36] GCC. 2020. GCC Bugzilla Main Page. <https://gcc.gnu.org/bugzilla/>.
- [37] GCC. 2022. 6.47.2 Extended Asm - Assembler Instructions with C Expression Operands. <https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>.
- [38] Greg Guest, Arwen Bunce, and Laura Johnson. 2006. How many interviews are enough? An experiment with data saturation and variability. *Field methods* 18, 1 (2006), 59–82.
- [39] Eli Friedman halayli. 2013. incorrect loop optimization at O2. https://bugs.llvm.org/show_bug.cgi?id=16602.
- [40] David Howells, Paul E. McKenney, Will Deacon, and Peter Zijlstra. 2020. LINUX KERNEL MEMORY BARRIERS. <https://www.kernel.org/doc/Documentation/memory-barriers.txt>.
- [41] Intel. 2018. Analyzing Potential Bounds Check Bypass Vulnerabilities. <https://software.intel.com/security-software-guidance/deep-dives/deep-dive-analyzing-potential-bounds-check-bypass-vulnerabilities>.
- [42] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. <https://software.intel.com/security-software-guidance/api-app/sites/default/files/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf>.
- [43] Daniel Borkmann James Yonan. 2013. crypto: crypto_memneq - add equality testing of memory regions w/o timing leaks. <https://github.com/torvalds/linux/commit/6bf37e5aa90f18baf5acf4874bca505dd667c37f>.
- [44] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. *ACM SIGPLAN Notices* 47, 6 (2012), 77–88.
- [45] Linus Torvalds Josh Poimboeuf. 2020. bitops: always inline sign extension helpers. <https://github.com/torvalds/linux/commit/f80ac98a641a03097cbc9fd4b6a41a8dd3b7ae>.
- [46] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.
- [47] Ingo Molnar Krzysztof Helt. 2008. x86: do not allow to optimize flag_is_changeable_p() (rev. 2). <https://github.com/torvalds/linux/commit/94f6bac1058fd59a8bd472d18c4b77f220d930b0>.
- [48] lcamtuf. 2020. american fuzzy lop. <https://lcamtuf.coredump.cx/afl/>.
- [49] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. *ACM SIGPLAN Notices* 49, 6 (2014), 216–226.
- [50] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized stress-testing of link-time optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 327–337.
- [51] Juneyoung Lee, Yoonseung Kim, Youngju Song, Chung-Kil Hur, Sanjoy Das, David Majnemer, John Regehr, and Nuno P. Lopes. 2017. Taming undefined behavior in LLVM. *ACM SIGPLAN Notices* 52, 6 (2017), 633–647. <https://doi.org/10.1145/3140587.3062343>
- [52] Xavier Leroy. 2007. Formal verification of an optimizing compiler. *Lecture Notes in Computer Science* 4533, 1 (2007).
- [53] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.
- [54] Xavier Leroy et al. 2012. The CompCert verified compiler.
- [55] Juanru Li, Zhiqiang Lin, Juan Caballero, Yuanyuan Zhang, and Dawu Gu. 2018. K-Hunt: Pinpointing insecure cryptographic keys from execution traces. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 412–425.
- [56] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F Donaldson. 2015. Many-core compiler fuzzing. *ACM SIGPLAN Notices* 50, 6 (2015), 65–76.
- [57] Jay P Lim, Vinod Ganapathy, and Santosh Nagarakatte. 2017. Compiler optimizations with retrofitting transformations: Is there a semantic mismatch?. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security*. 37–42.
- [58] Linux. 2020. PROPER CARE AND FEEDING OF RETURN VALUES FROM rcu_dereference(). https://www.kernel.org/doc/Documentation/RCU/rcu_dereference.rst.
- [59] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 973–990.
- [60] Changming Liu, Yaohui Chen, and Long Lu. 2021. KUBO: Precise and Scalable Detection of User-triggerable Undefined Behavior Bugs in OS Kernel. In *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS’21)*.
- [61] LLVM. 2020. Clang 12 documentation: SafeStack. <https://clang.llvm.org/docs/SafeStack.html>.
- [62] LLVM. 2020. LLVM Bugzilla Main Page. <https://bugs.llvm.org/>.
- [63] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 65–79.
- [64] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1769–1786.
- [65] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. Unisan: Proactive kernel memory initialization to eliminate data leaks. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 920–932.
- [66] Daniel Borkmann mancha security and Herbert Xu. 2015. lib: memzero_explicit: use barrier instead of OPTIMIZER_HIDE_VAR. <https://github.com/torvalds/linux/commit/0b053c9518292705736329a8fe20ef4686ffc8e9>.

- [67] William Mansky and Elsa Gunter. 2010. A framework for formal verification of compiler optimizations. In *International Conference on Interactive Theorem Proving*. Springer, 371–386.
- [68] last modified by David Svoboda Martin Sebor. 2018. Undefined Behavior. <https://wiki.sei.cmu.edu/confluence/display/c/CC.+Undefined+Behavior>.
- [69] John McCarthy and James Painter. 1967. Correctness of a compiler for arithmetic expressions. *Mathematical Aspects of Computer Science 1* (1967).
- [70] David S. Miller and Linus Torvalds. 2007. [PATCH] video/aty/mach64_ct.c: fix bogus delay loop. <https://github.com/torvalds/linux/commit/8690ba446defe2e2b81803756c099d2943dfd5fd>.
- [71] MITRE. 2008. CVE-2008-1685. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=2008-1685>.
- [72] MITRE. 2019. CVE-2019-1010006. <https://cve.mitre.org/cgi-bin/cvename.cgi?name=2019-1010006>.
- [73] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 919–936.
- [74] Stephan Mueller. 2015. lib: make memzero_explicit more robust against dead store elimination. <https://github.com/torvalds/linux/commit/7829fb09a2b4268b30dd9bc782fa5ebee278b137>.
- [75] Paul Muntean, Matthias Fischer, Gang Tan, Zhiqiang Lin, Jens Grossklags, and Claudia Eckert. 2018. τ CFI: Type-Assisted Control Flow Integrity for x86-64 Binaries. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 423–444.
- [76] Kazuhiro Nakamura and Nagisa Ishiura. 2016. Random testing of C compilers based on test program generation by equivalence transformation. In *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 676–679.
- [77] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 25–34.
- [78] Russell King Nicolas Pitre. 2008. [ARM] 5196/1: fix inline asm constraints for preload. <https://github.com/torvalds/linux/commit/16f719de62809e224e37c320760c3ce59098d862>.
- [79] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1481–1498. <https://www.usenix.org/conference/usenixsecurity20/presentation/oleksenko>
- [80] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal approaches to secure compilation: A survey of fully abstract compilation and related work. *ACM Computing Surveys (CSUR)* 51, 6 (2019), 1–36.
- [81] Marco Patrignani and Deepak Garg. 2017. Secure compilation and hyperproperty preservation. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 392–404.
- [82] Marco Patrignani and Marco Guarnieri. 2021. Exorcising spectres with secure compilers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 445–461.
- [83] Mathias Payer, Antonio Barresi, and Thomas R Gross. 2014. Lockdown: Dynamic control-flow integrity. *arXiv preprint arXiv:1407.0549* (2014).
- [84] Hui Peng, Yan Shoshitaishvili, and Mathias Payer. 2018. T-Fuzz: fuzzing by program transformation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 697–710.
- [85] Richard Biener Peter Horton. 2007. gcc-4.2.2 generates bad code on ARM. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=33949.
- [86] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. 2021. SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets.. In *NDSS*.
- [87] Abhay Rathi. 2017. Integer Promotions in C. <https://www.geeksforgeeks.org/integer-promotions-in-c/>.
- [88] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 335–346.
- [89] Alan Romano, Xinyue Liu, Yonghwi Kwon, and Weihang Wang. 2021. An Empirical Study of Bugs in WebAssembly Compilers. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–54.
- [90] Guang Gong Rong Jian, Leecraso. 2021. Put in one bug and pop out more: An effective way of bug hunting in Chrome. <https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Leecraso-Put-In-One-Bug-And-Pop-Out-More-An-Effective-Way-Of-Bug-Hunting-In-Chrome.pdf>.
- [91] Mancha security and Daniel Borkmann. 2015. lib: memzero_explicit: use barrier instead of OPTIMIZER_HIDE_VAR. <https://github.com/torvalds/linux/commit/0b053c9518292705736329a8fe20ef4686ffc8e9>.
- [92] Konstantin Serebryany and Timur Iskhodzhanov. 2009. ThreadSanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*. 62–71.
- [93] Laurent Simon, David Chisnall, and Ross Anderson. 2018. What You Get is What You C: Controlling Side Effects in Mainstream C Compilers. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. <https://doi.org/10.1109/eurosp.2018.00009>
- [94] Rohit Sinha, Sriram Rajamani, and Sanjit A Seshia. 2017. A compiler and verifier for page access oblivious computation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 649–660.
- [95] Yevheniy Soloshenko. 2018. Condition check is optimized out for volatile unsigned char / short. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=87060.
- [96] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. 2019. SoK: sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1275–1295.
- [97] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding and analyzing compiler warning defects. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 203–213.
- [98] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. 2016. Toward understanding compiler bugs in GCC and LLVM. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 294–305.
- [99] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical software engineering* 19, 6 (2014), 1665–1705.
- [100] Mikko Tiuhonen. 2004. GCC Bug 15685. https://gcc.gnu.org/bugzilla/show_bug.cgi?id=15685.
- [101] Jean Tourrilhes. 2003. Invalid compilation without -fno-strict-aliasing. <https://lkm1.org/lkml/2003/2/25/270>.
- [102] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.
- [103] Ilja van Sprundel. 2018. Memsad: why clearing memory is hard. <https://media.ccc.de/v/35c3-9788-memsad>.
- [104] Daniel Votipka, Eric Zhang, and Michelle L. Mazurek. 2021. HackEd: A Pedagogical Analysis of Online Vulnerability Discovery Exercises.

- In 2021 *IEEE Symposium on Security and Privacy (SP)*. 1268–1285. <https://doi.org/10.1109/SP40001.2021.00092>
- [105] Son Tuan Vu, Albert Cohen, Karine Heydemann, Arnaud de Grandmaison, and Christophe Guillon. 2021. Secure Optimization Through Opaque Observations. *arXiv preprint arXiv:2101.06039* (2021).
- [106] Son Tuan Vu, Karine Heydemann, Arnaud de Grandmaison, and Albert Cohen. 2020. Secure Delivery of Program Properties through Optimizing Compilation. In *Proceedings of the 29th International Conference on Compiler Construction* (San Diego, CA, USA) (*CC 2020*). Association for Computing Machinery, New York, NY, USA, 14–26. <https://doi.org/10.1145/3377555.3377897>
- [107] Dmitriy Vyukov. 2013. Benign Data Races: What Could Possibly Go Wrong? <https://software.intel.com/content/www/us/en/develop/blogs/benign-data-races-what-could-possibly-go-wrong.html>.
- [108] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. Undefined behavior: What Happened to My Code?. In *Proceedings of the Asia-Pacific Workshop on Systems - APSYS '12*. ACM Press. <https://doi.org/10.1145/2349896.2349905>
- [109] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 260–275.
- [110] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. 2016. A Differential Approach to Undefined Behavior Detection. *Communications of the Acm* 59, 3 (2016), 99–106. <https://doi.org/10.1145/2885256>
- [111] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. 2019. Ct-wasm: type-driven secure cryptography for the web ecosystem. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.
- [112] Jinpeng Wei and Calton Pu. 2005. TOCTTOU Vulnerabilities in UNIX-style File Systems: An Anatomical Study. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4 (FAST'05)*.
- [113] Nick Wilfahrt. 2016. Dirty COW (CVE-2016-5195) is a privilege escalation vulnerability in the Linux Kernel. <https://dirtycow.ninja/>.
- [114] Zekai Wu, Wei Liu, Mingyue Liang, and Kai Song. 2020. Finding Bugs Compiler Knows but Doesn't Tell You: Dissecting Undefined Behavior Optimizations in LLVM. <https://i.blackhat.com/eu-20/Wednesday/eu-20-Wu-Finding-Bugs-Compiler-Knows-But-Does-Not-Tell-You-Dissecting-Undefined-Behavior-Optimizations-In-LLVM.pdf>.
- [115] Song Liu Xiao Ni. 2021. md: Set prev_flush_start and flush_bio in an atomic way. <https://github.com/torvalds/linux/commit/dc5d17a3c39b06aef866afca19245a9c9fb533a79>.
- [116] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. 2018. Precise and Scalable Detection of Double-Fetch Bugs in OS Kernels. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
- [117] Debin Gao Yan Lin. 2021. When Function Signature Recovery Meets Compiler Optimization. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (Oakland)*. San Francisco, CA.
- [118] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.
- [119] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. 2017. Dead store elimination (still) considered harmful. In *26th {USENIX} Security Symposium*. 1025–1040.
- [120] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 129–140.
- [121] Peter Zijlstra and Tejun Heo. 2012. lockdep: fix oops in processing workqueue. <https://github.com/torvalds/linux/commit/4d82a1debbffec129cc387aaafa8f40b7bbab3297>.

12 Appendix

12.1 User study materials

12.1.1 Recruitment Requirements

Participants who meet the following requirement are selected to participate the user study:

- have C programming experience > 2 years OR knew about Undefined Behavior.

For graduate students, in addition to the above requirements, they also need to

- have a background in software security or programming language;
- have participated in at least a C project as the main developer (contributing > 500 LoC).

12.1.2 Online Consent Form

We provide an online consent form for participants to read before agreeing to be in the study. It includes the purpose, the procedure of this study, and the risks and benefits of taking part in this study. An anonymous copy of it can be viewed at https://docs.google.com/forms/d/e/1FAIpQLSexFB91zcGK80JUFU_2xqKstNjXtz-gWPH990nODihw9ZF6NQ/viewform?usp=sf_link.

12.1.3 Background Survey

The background survey is used to record demographic information. It includes questions of the job and C language experience (in years) of participants.

12.1.4 Online Questionnaire

An anonymous copy of our online questionnaire can be viewed at https://docs.google.com/forms/d/e/1FAIpQLSc1EagB7LyiSfjdg-nl1C4TBrpr5zVN9Z8P3VufBRQK005_AQ/viewform?usp=sf_link.

12.2 More details of the bug collection study

The keywords we used in our filtering policy (keyword searching and intersection) of the bug collection study for Linux patch history and their corresponding Regular Expression can be viewed in [Table 8](#).

Table 9: The compiler options of our selected prevention strategies.

Strategy	Compiler options
O3	gcc -O3
	clang -O3
O2	gcc -O2
	clang -O2
O1	gcc -O1
	clang -O1
O0	gcc -O0
	clang -O0
All-UB	gcc -O3 -fno-strict-overflow -fwrapv -fno-delete-null-pointer-checks -fno-strict-aliasing -fno-aggressive-loop-optimizations
	clang -O3 -fno-strict-overflow -fno-delete-null-pointer-checks -fno-strict-aliasing -fwrapv
All-CISB	gcc -O3 -fno-tree-dominator-opts -fno-tree-vrp -fno-tree-fre -fno-strict-overflow -fno-dce -fno-tree-ccp -fno-tree-copy-prop -fno-tree-forwprop -fno-tree-ter -fno-tree-pre -fno-strict-aliasing -fno-aggressive-loop-optimizations -fno-builtin -fno-tree-dse -fno-optimize-strlen -fno-tree-dce -fno-cse-follow-jumps -fno-unswitch-loops
	clang -O3 -fno-builtin -fno-strict-overflow -fno-strict-aliasin -fwrapv -fno-delete-null-pointer-checks
UBSan	gcc -O3 -fsanitize=undefined
	clang -O3 -fsanitize=undefined
Wall	gcc -O3 -Wall
	clang -O3 -Wall

Table 10: Some compiler options and the assumptions/optimizations they can block.

Options	Assumptions/Optimizations
-fno-builtin	Optimization of builtin function
-ffreestanding	Optimization of builtin function
-fno-delete-null-pointer-checks	Used pointers cannot be NULL
-fwrapv	No integer overflow
-fno-strict-overflow	No integer overflow
-fwrapv-pointer	No pointer overflow
-fno-strict-aliasing	No violating strict aliasing
-fno-aggressive-loop-optimizations	Aggressive loop optimization
-fno-allow-store-data-races	Introduce data race on stores

Table 8: The short name of keywords we used in our study and their corresponding Regular Expression.

Keyword	Regular Expression
gcc	[Gg]cc GCC
clang	[Cc]lang CLANG
compiler	[Cc]ompiler
optimi	[Oo]ptimi[sz][ae]
security	[Ss]ecurity [Dd]anger
ub	[Uu]ndefined behavior
side channel	[Ss]ide channel
attack	[Aa]ttack
race	[Rr]ace condition
compiler assum	[Cc]ompiler assum [Gg]cc assum GCC assum [Cc]lang assum CLANG assum
(cc+opti).*break	[Oo]ptimi[sz][ae].*break [Gg]cc.*break GCC.*break [Cc]lang.*break CLANG.*break
(cc+opti).*introduc	[Oo]ptimi[sz][ae].*introduc [Gg]cc.*introduc GCC.*introduc [Cc]lang.*introduc CLANG.*introduc
struct padding	[Ss]truct [Pp]adding
information leak	[Ii]nformation [Ll]eak
reorder	[Rr]eorder

12.3 More details of current mitigations

The compiler has provided three general mitigations to prevent *CISB*, namely optimization options for preventing bugs, runtime sanitizers and compilation time warnings. We select different compiler mitigation strategies and study their effectiveness and performance, as listed in Table 6. Here we show the corresponding compiler options of these strategies in Table 9.

We also present the compiler options and the *No-UB* assumptions/optimizations they can block in Table 10.