Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior

Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama
MIT CSAIL

Abstract

This paper studies an emerging class of software bugs called *optimization-unstable code*: code that is unexpectedly discarded by compiler optimizations due to undefined behavior in the program. Unstable code is present in many systems, including the Linux kernel and the Postgres database. The consequences of unstable code range from incorrect functionality to missing security checks.

To reason about unstable code, this paper proposes a novel model, which views unstable code in terms of optimizations that leverage undefined behavior. Using this model, we introduce a new static checker called STACK that precisely identifies unstable code. Applying STACK to widely used systems has uncovered 160 new bugs that have been confirmed and fixed by developers.

1 Introduction

The specifications of many programming languages designate certain code fragments as having *undefined behavior* [15: §2.3], giving compilers the freedom to generate instructions that behave in arbitrary ways in those cases. For example, in C the "use of a nonportable or erroneous program construct or of erroneous data" leads to undefined behavior [24: §3.4.3].

One way in which compilers exploit undefined behavior is to optimize a program under the assumption that the program never invokes undefined behavior. A consequence of such optimizations is especially surprising to many programmers: code which works with optimizations turned off (e.g., `-O0`) breaks with a higher optimization level (e.g., `-O2`), because the compiler considers part of the code dead and discards it. We call such code *optimization-unstable code*, or just unstable code for short. If the discarded unstable code happens to be used for security checks, the optimized system will become vulnerable to attacks.

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522728

```
char *buf = ...;
char *buf_end = ...;
unsigned int len = ...;
if (buf + len >= buf_end)
    return;   /* len too large */
if (buf + len < buf)
    return;   /* overflow, buf+len wrapped around */
/* write to buf[0..len-1] */
```

Figure 1: A pointer overflow check found in several code bases. The code becomes vulnerable as gcc optimizes away the second `if` statement [13].

This paper presents the first systematic approach for reasoning about and detecting unstable code. We implement this approach in a static checker called STACK, and use it to show that unstable code is present in a wide range of systems software, including the Linux kernel and the Postgres database. We estimate that unstable code exists in 40% of the 8,575 Debian Wheezy packages that contain C/C++ code. We also show that compilers are increasingly taking advantage of undefined behavior for optimizations, leading to more vulnerabilities related to unstable code.

To understand unstable code, consider the pointer overflow check `buf + len < buf` shown in Figure 1, where `buf` is a pointer and `len` is a positive integer. The programmer's intention is to catch the case when `len` is so large that `buf + len` wraps around and bypasses the first check in Figure 1. We have found similar checks in a number of systems, including the Chromium browser [7], the Linux kernel [49], and the Python interpreter [37].

While this check appears to work with a flat address space, it fails on a segmented architecture [23: §6.3.2.3]. Therefore, the C standard states that an overflowed pointer is undefined [24: §6.5.6/p8], which allows gcc to simply assume that no pointer overflow ever occurs on *any* architecture. Under this assumption, `buf + len` must be larger than `buf`, and thus the "overflow" check always evaluates to *false*. Consequently, gcc removes the check, paving the way for an attack to the system [13].

In addition to introducing new vulnerabilities, unstable code can amplify existing weakness in the system. Figure 2 shows a mild defect in the Linux kernel, where the programmer incorrectly placed the dereference `tun->sk`

```
struct tun_struct *tun = ...;
struct sock *sk = tun->sk;
if (!tun)
    return POLLERR;
/* write to address based on tun */
```

Figure 2: A null pointer dereference vulnerability (CVE-2009-1897) in the Linux kernel, where the dereference of pointer `tun` is before the null pointer check. The code becomes exploitable as gcc optimizes away the null pointer check [10].

before the null pointer check `!tun`. Normally, the kernel forbids access to page zero; a null `tun` pointing to page zero causes a kernel oops at `tun->sk` and terminates the current process. Even if page zero is made accessible (e.g., via `mmap` or some other exploits [25, 45]), the check `!tun` would catch a null `tun` and prevent any further exploits. In either case, an adversary should *not* be able to go beyond the null pointer check.

Unfortunately, unstable code can turn this simple bug into an exploitable vulnerability. For example, when gcc first sees the dereference `tun->sk`, it concludes that the pointer `tun` must be non-null, because the C standard states that dereferencing a null pointer is undefined [24: §6.5.3]. Since `tun` is non-null, gcc further determines that the null pointer check is unnecessary and eliminates the check, making a privilege escalation exploit possible that would not otherwise be [10].

Poor understanding of unstable code is a major obstacle to reasoning about system behavior. For programmers, compilers that discard unstable code are often "baffling" and "make no sense" [46], merely gcc's "creative reinterpretation of basic C semantics" [27]. On the other hand, compiler writers argue that the C standard allows such optimizations, which many compilers exploit (see §2.3); it is the "broken code" [17] that programmers should fix.

Who is right in this debate? From the compiler's point of view, the programmers made a mistake in their code. For example, Figure 2 clearly contains a bug, and even Figure 1 is arguably incorrect given a strict interpretation of the C standard. However, these bugs are quite subtle, and understanding them requires detailed knowledge of the language specification. Thus, it is not surprising that such bugs continue to proliferate.

From the programmer's point of view, the compilers are being too aggressive with their optimizations. However, optimizations are important for achieving good performance; many optimizations fundamentally rely on the precise semantics of the C language, such as eliminating needless null pointer checks or optimizing integer loop variables [20, 29]. Thus, it is difficult for compiler writers to distinguish legal yet complex optimizations from an optimization that goes too far and violates the programmer's intent [29: §3].

This paper helps resolve this debate by introducing a model for identifying unstable code that allows a compiler to generate precise warnings when it removes code based on undefined behavior. The model specifies precise conditions under which a code fragment can induce undefined behavior. Using these conditions we can identify fragments that can be eliminated under the assumption that undefined behavior is never triggered; specifically, any fragment that is reachable only by inputs that trigger undefined behavior is unstable code. We make this model more precise in §3.

The Stack checker implements this model to identify unstable code. For the example in Figure 2, it emits a warning that the null pointer check `!tun` is unstable due to the earlier dereference `tun->sk`. Stack first computes the undefined behavior conditions for a wide range of constructs, including pointer and integer arithmetic, memory access, and library function calls. It then uses a constraint solver [3] to determine whether the code can be simplified away given the undefined behavior conditions, such as whether the code is reachable only when the undefined behavior conditions are *true*. We hope that Stack will help programmers find unstable code in their applications, and that our model will help compilers make better decisions about what optimizations might be unsafe and when an optimizer should produce a warning.

We implemented the Stack checker using the LLVM compiler framework [30] and the Boolector solver [3]. Applying it to a wide range of systems uncovered 160 new bugs, which were confirmed and fixed by the developers. We also received positive feedback from outside users who, with the help of Stack, fixed additional bugs in both open-source and commercial code bases. Our experience shows that unstable code is a widespread threat in systems, that an adversary can exploit vulnerabilities caused by unstable code with major compilers, and that Stack is useful for identifying unstable code.

The main contributions of this paper are:

- a new model for understanding unstable code,
- a static checker for identifying unstable code, and
- a detailed case study of unstable code in real systems.

Another conclusion one can draw from this paper is that language designers should be careful with defining language construct as undefined behavior. Almost every language allows a developer to write programs that have undefined meaning according to the language specification. Our experience with C/C++ indicates that being liberal with what is undefined can lead to subtle bugs.

The rest of the paper is organized as follows. §2 provides background information. §3 presents our model of unstable code. §4 outlines the design of Stack. §5 summarizes its implementation. §6 reports our experience of applying Stack to identify unstable code and evaluates Stack's techniques. §7 covers related work. §8 concludes.

	Construct	Sufficient condition	Undefined behavior
Language	$p + x$	$p_\infty + x_\infty \notin [0, 2^n - 1]$	pointer overflow
	$*p$	$p = \texttt{NULL}$	null pointer dereference
	$x \text{ op}_s y$	$x_\infty \text{ op}_s y_\infty \notin [-2^{n-1}, 2^{n-1} - 1]$	signed integer overflow
	x / y, x % y	$y = 0$	division by zero
	$x << y$, $x >> y$	$y < 0 \vee y \geq n$	oversized shift
	$a[x]$	$x < 0 \vee x \geq \texttt{ARRAY\_SIZE}(a)$	buffer overflow
Library	$\texttt{abs}(x)$	$x = -2^{n-1}$	absolute value overflow
	$\texttt{memcpy}(\texttt{dst}, \texttt{src}, \texttt{len})$	$\lvert\texttt{dst} - \texttt{src}\rvert < \texttt{len}$	overlapping memory copy
	use q after $\texttt{free}(p)$	$\text{alias}(p, q)$	use after free
	use q after $p' := \texttt{realloc}(p, ...)$	$\text{alias}(p, q) \wedge p' \neq \texttt{NULL}$	use after realloc

Figure 3: A list of sufficient (though not necessary) conditions for undefined behavior in certain C constructs [24: §J.2]. Here p, p', q are n-bit pointers; x, y are n-bit integers; a is an array, the capacity of which is denoted as $\texttt{ARRAY\_SIZE}(a)$; op_s refers to binary operators +, -, *, /, % over signed integers; x_∞ means to consider x as infinitely ranged; NULL is the null pointer; $\text{alias}(p, q)$ predicates whether p and q point to the same object.

2 Background

This section provides some background on undefined behavior and how it can lead to unstable code. It builds on earlier surveys [26, 41, 49] and blog posts [29, 39, 40] that describe unstable code examples, and extends them by investigating the evolution of optimizations in compilers.

2.1 Undefined behavior

Figure 3 shows a list of constructs and their undefined behavior conditions, as specified in the C standard [24: §J.2]. One category of undefined behavior is simply programming errors, such as null pointer dereference, buffer overflow, and use after free. The other category is non-portable constructs, the hardware implementations of which often have subtle differences.

For instance, when signed integer overflow or division by zero occurs, a division instruction traps on x86 [22: §3.2], while it silently produces an undefined result on PowerPC [21: §3.3.8]. Another example is shift instructions: left-shifting a 32-bit one by 32 bits produces 0 on ARM and PowerPC, but 1 on x86; however, left-shifting a 32-bit one by 64 bits produces 0 on ARM, but 1 on x86 and PowerPC. Wang et al.'s survey [49] provides more details of such architectural differences.

To build a portable system, the language standard could impose uniform behavior over erroneous or non-portable constructs, as many higher-level languages do. In doing so, the compiler would have to synthesize extra instructions. For example, to enforce well-defined error handling (e.g., run-time exception) on buffer overflow, the compiler would need to insert extra bounds checks for memory access operations. Similarly, to enforce a consistent shift behavior on x86, for every $x << y$ the compiler would need to insert a check against y (unless it is able to prove that y is not oversized), as follows:

$$\text{if } (y < 0 \vee y \geq n) \text{ then } 0 \text{ else } x << y.$$

The C-family languages employ a different approach. Aiming for system programming, their specifications choose to trust programmers [23: §0] and assume that their code will never invoke undefined behavior. This assumption gives more freedom to the compiler than simply saying that the result of a particular operation is architecture-dependent. While it allows the compiler to generate efficient code without extra checks, the assumption also opens the door to unstable code.

2.2 Examples of unstable code

The top row of Figure 4 shows six representative examples of unstable code in the form of sanity checks. All of these checks may evaluate to *false* and become dead code under optimizations, even though none appear to directly invoke undefined behavior. We will use them to test existing compilers in §2.3.

The check $p + 100 < p$ resembles Figure 1, which is dead assuming no pointer overflow.

The null pointer check $!p$ with an earlier dereference is from Figure 2, which is dead assuming no null pointer dereference.

The check $x + 100 < x$ with a signed integer x becomes dead assuming no signed integer overflow. It once led to a harsh debate between some C programmers and gcc developers [17].

Another check $x^+ + 100 < 0$ tests whether optimizations perform more elaborate reasoning assuming no signed integer overflow; x^+ is known to be positive.

The shift check $!(1 << x)$ was intended to catch a large x, from a patch to the ext4 file system [31]. It becomes dead assuming no oversized shift amount.

The check $\texttt{abs}(x) < 0$ was used in the PHP interpreter to catch the most negative value (i.e., -2^{n-1}). It becomes dead when optimizations understand this library function and assume no absolute value overflow [18].

	`if` $(p + 100 < p)$	$*p$; `if` $(!p)$	`if` $(x + 100 < x)$	`if` $(x^+ + 100 < 0)$	`if` $(!(1 << x))$	`if` (`abs`$(x) < 0$)
gcc-2.95.3	–	–	`O1`	–	–	–
gcc-3.4.6	–	`O2`	`O1`	–	–	–
gcc-4.2.1	`O0`	–	`O2`	–	–	`O2`
gcc-4.8.1	`O2`	`O2`	`O2`	`O2`	–	`O2`
clang-1.0	`O1`	–	–	–	–	–
clang-3.3	`O1`	–	`O1`	–	`O1`	–
aCC-6.25	–	–	–	–	–	`O3`
armcc-5.02	–	–	`O2`	–	–	–
icc-14.0.0	–	`O2`	`O1`	`O2`	–	–
msvc-11.0	–	`O1`	–	–	–	–
open64-4.5.2	`O1`	–	`O2`	–	–	`O2`
pathcc-1.0.0	`O1`	–	`O2`	–	–	`O2`
suncc-5.12	–	`O3`	–	–	–	–
ti-7.4.2	`O0`	–	`O0`	`O2`	–	–
windriver-5.9.2	–	–	`O0`	–	–	–
xlc-12.1	`O3`	–	–	–	–	–

Figure 4: Optimizations of unstable code in popular compilers: gcc, clang, aCC, armcc, icc, msvc, open64, pathcc, suncc, TI's TMS320C6000, Wind River's Diab compiler, and IBM's XL C compiler. In the examples, p is a pointer, x is a signed integer, and x^+ is a positive signed integer. In each cell, "`O`n" means that the specific version of the compiler optimizes the check into *false* and discards it at optimization level n, while "–" means that the compiler does not discard the check at any level.

2.3 An evolution of optimizations

We chose 12 well-known C/C++ compilers to see what they do with the unstable code examples: two open-source compilers (gcc and clang) and ten recent commercial compilers (HP's aCC, ARM's armcc, Intel's icc, Microsoft's msvc, AMD's open64, PathScale's pathcc, Oracle's suncc, TI's TMS320C6000, Wind River's Diab compiler, and IBM's XL C compiler). For every unstable code example, we test whether a compiler optimizes the check into *false*, and if so, find the lowest optimization level `-O`n at which it happens. The result is shown in Figure 4.

We further use gcc and clang to study the evolution of optimizations, as the history is easily accessible. For gcc, we chose the following representative versions that span more than a decade:

- gcc 2.95.3, the last 2.x, released in 2001;
- gcc 3.4.6, the last 3.x, released in 2006;
- gcc 4.2.1, the last GPLv2 version, released in 2007 and still widely used in BSD systems;
- gcc 4.8.1, the latest version, released in 2013.

For comparison, we chose two versions of clang, 1.0 released in 2009, and the latest 3.3 released in 2013.

We make the following observations of existing compilers from Figure 4. First, discarding unstable code is common among compilers, not just in recent gcc versions as some programmers have claimed [27]. Even gcc 2.95.3 eliminates $x + 100 < x$. Some compilers discard unstable code that gcc does not (e.g., clang on $1 << x$).

Second, from different versions of gcc and clang, we see more unstable code discarded as the compilers evolve to adopt new optimizations. For example, gcc 4.x is more aggressive in discarding unstable code compared to gcc 2.x, as it uses a new value range analysis [36].

Third, discarding unstable code occurs with standard optimization options, mostly at `-O2`, the default optimization level for release build (e.g., `autoconf` [32: §5.10.3]); some compilers even discard unstable code at the lowest level of optimization `-O0`. Hence, lowering the optimization level as Postgres did [28] is an unreliable way of working around unstable code.

Fourth, optimizations exploit undefined behavior not only from the core language features, but also from library functions (e.g., `abs` [18] and `realloc` [40]) as the compilers evolve to understand them.

As compilers improve their optimizations, for example, by implementing new algorithms (e.g., gcc 4.x's value range analysis) or by exploiting undefined behavior from more constructs (e.g., library functions), we anticipate an increase in bugs due to unstable code.

3 Model for unstable code

Discarding unstable code, as the compilers surveyed in §2 do, is legal as per the language standard, and thus is *not* a compiler bug [39: §3]. But, it is baffling to programmers. Our goal is to identify such unstable code fragments and generate warnings for them. As we will see in §6.2, these warnings often identify code that programmers want to fix, instead of having the compiler remove it silently. This goal requires a precise model for understanding unstable code so as to generate warnings only for code that is unstable, and not for code that is trivially dead and can be safely removed. This section introduces a model for thinking about unstable code and a framework with two algorithms for identifying it.

3.1 Unstable code

To formalize a programmer's misunderstanding of the C specification that leads to unstable code, let C* denote a C dialect that assigns well-defined semantics to code fragments that have undefined behavior in C. For example, C* is defined for a flat address space, a null pointer that maps to address zero, and wrap-around semantics for pointer and integer arithmetic [38]. A code fragment e is a statement or expression at a particular source location in program $\mathcal{P}$. If the compiler can transform the fragment e in a way that would change $\mathcal{P}$'s behavior under C* but not under C, then e is unstable code.

Let $\mathcal{P}[e/e']$ be a program formed by replacing e with some fragment e' at the same source location. When is it legal for a compiler to transform $\mathcal{P}$ into $\mathcal{P}[e/e']$, denoted as $\mathcal{P} \rightsquigarrow \mathcal{P}[e/e']$? In a language specification without undefined behavior, the answer is straightforward: it is legal if for every input, both $\mathcal{P}$ and $\mathcal{P}[e/e']$ produce the same result. In a language specification *with* undefined behavior, the answer is more complicated; namely, it is legal if for every input, one of the following is true:

- both $\mathcal{P}$ and $\mathcal{P}[e/e']$ produce the same results without invoking undefined behavior, or
- $\mathcal{P}$ invokes undefined behavior, in which case it does not matter what $\mathcal{P}[e/e']$ does.

Using this notation, we define unstable code below.

Definition 1 (Unstable code)**.** A code fragment e in program $\mathcal{P}$ is unstable *w.r.t.* language specifications C and C* iff there exists a fragment e' such that $\mathcal{P} \rightsquigarrow \mathcal{P}[e/e']$ is legal under C but *not* under C*.

For example, for the sanity checks listed in Figure 4, a C compiler is entitled to replace them with *false*, as this is legal according to the C specification, whereas a hypothetical C* compiler cannot do the same. Therefore, these checks are unstable code.

3.2 Approach for identifying unstable code

The above definition captures what unstable code is, but does not provide a way of finding unstable code, because it is difficult to reason about how an entire program will behave. As a proxy for a change in program behavior, Stack looks for code that can be transformed by some optimizer O under C but not under C*. In particular, Stack does this using a two-phase scheme:

1. run O without taking advantage of undefined behavior, which resembles optimizations under C*; and
2. run O again, this time taking advantage of undefined behavior, which resembles (more aggressive) optimizations under C.

If O optimizes extra code in the second phase, we assume the reason O did not do so in the first phase is because it would have changed the program's semantics under C*, and so Stack considers that code to be unstable.

Stack's optimizer-based approach to finding unstable code will miss unstable code that a specific optimizer cannot eliminate in the second phase, even if there exists some optimizer that could. This approach will also generate false reports if the optimizer is not aggressive enough in eliminating code in the first phase. Thus, one challenge in Stack's design is coming up with an optimizer that is sufficiently aggressive to minimize these problems.

In order for this approach to work, Stack requires an optimizer that can selectively take advantage of undefined behavior. To build such optimizers, we formalize what it means to "take advantage of undefined behavior" in §3.2.1, by introducing the *well-defined program assumption*, which captures C's assumption that programmers never write programs that invoke undefined behavior. Given an optimizer that can take explicit assumptions as input, Stack can turn on (or off) optimizations based on undefined behavior by supplying (or not) the well-defined program assumption to the optimizer. We build two aggressive optimizers that follow this approach: one that eliminates unreachable code (§3.2.2) and one that simplifies unnecessary computation (§3.2.3).

3.2.1 Well-defined program assumption

We formalize what it means to take advantage of undefined behavior in an optimizer as follows. Consider a program with input $\mathbf{x}$. Given a code fragment e, let $R_e(\mathbf{x})$ denote its *reachability condition*, which is *true* iff e will execute under input $\mathbf{x}$; and let $U_e(\mathbf{x})$ denote its *undefined behavior condition*, or UB condition for short, which indicates whether e exhibits undefined behavior on input $\mathbf{x}$, assuming C semantics (see Figure 3).

Both $R_e(\mathbf{x})$ and $U_e(\mathbf{x})$ are boolean expressions. For example, given a pointer dereference $*\mathtt{p}$ in expression e, one UB condition $U_e(\mathbf{x})$ is $p = \mathtt{NULL}$ (i.e., causing a null pointer dereference).

Intuitively, in a well-defined program to dereference pointer p, p must be non-null. In other words, the negation of its UB condition, $p \neq \mathtt{NULL}$, must hold whenever the expression executes. We generalize this below.

Definition 2 (Well-defined program assumption)**.** A code fragment e is well-defined on an input $\mathbf{x}$ iff executing e never triggers undefined behavior at e:

$$R_e(\mathbf{x}) \rightarrow \neg U_e(\mathbf{x}). \tag{1}$$

Furthermore, a program is well-defined on an input iff every fragment of the program is well-defined on that input, denoted as Δ:

$$\Delta(\mathbf{x}) = \bigwedge_{e \in \mathcal{P}} R_e(\mathbf{x}) \rightarrow \neg U_e(\mathbf{x}). \tag{2}$$

```
1: procedure ELIMINATE(P)
2:     for all e ∈ P do
3:         if R_e(x) is UNSAT then
4:             REMOVE(e)                          ▷ trivially unreachable
5:         else
6:             if R_e(x) ∧ Δ(x) is UNSAT then
7:                 REPORT(e)
8:                 REMOVE(e)                      ▷ unstable code eliminated
```

Figure 5: The elimination algorithm. It reports unstable code that becomes unreachable with the well-defined program assumption.

3.2.2 Eliminating unreachable code

The first algorithm identifies unstable statements that can be eliminated (i.e., $\mathcal{P} \rightsquigarrow \mathcal{P}[e/\varnothing]$ where e is a statement). For example, if reaching a statement requires triggering undefined behavior, then that statement must be unreachable. We formalize this below.

Theorem 1 (Elimination)**.** In a well-defined program $\mathcal{P}$, an optimizer can eliminate code fragment e, if there is no input $\mathbf{x}$ that both reaches e and satisfies the well-defined program assumption $\Delta(\mathbf{x})$:

$$\nexists \mathbf{x}\colon R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}). \tag{3}$$

The boolean expression $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ is referred as the *elimination query*.

Proof. Assuming $\Delta(\mathbf{x})$ is *true*, if the elimination query $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ always evaluates to *false*, then $R_e(\mathbf{x})$ must be *false*, meaning that e must be unreachable. One can then safely eliminate e. □

Consider Figure 2 as an example. There is one input `tun` in this program. To pass the earlier `if` check, the reachability condition of the `return` statement is `!tun`. There is one UB condition `tun = NULL`, from the pointer dereference `tun->sk`, the reachability condition of which is *true*. As a result, the elimination query $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ for the `return` statement is:

$$\texttt{!tun} \wedge (\mathit{true} \rightarrow \neg(\texttt{tun} = \texttt{NULL})).$$

Clearly, there is no `tun` that satisfies this query. Therefore, one can eliminate the `return` statement.

With the above definition it is easy to construct an algorithm to identify unstable due to code elimination (see Figure 5). The algorithm first removes unreachable fragments without the well-defined program assumption, and then warns against fragments that become unreachable with this assumption. The latter are unstable code.

3.2.3 Simplifying unnecessary computation

The second algorithm identifies unstable expressions that can be optimized into a simpler form (i.e., $\mathcal{P} \rightsquigarrow \mathcal{P}[e/e']$ where e and e' are expressions). For example, if evaluating a boolean expression to *true* requires triggering undefined behavior, then that expression must evaluate to *false*. We formalize this below.

```
1: procedure SIMPLIFY(P, oracle)
2:     for all e ∈ P do
3:         for all e' ∈ PROPOSE(oracle, e) do
4:             if e(x) ≠ e'(x) ∧ R_e(x) is UNSAT then
5:                 REPLACE(e, e')
6:                 break                          ▷ trivially simplified
7:             if e(x) ≠ e'(x) ∧ R_e(x) ∧ Δ(x) is UNSAT then
8:                 REPORT(e)
9:                 REPLACE(e, e')
10:                break                          ▷ unstable code simplified
```

Figure 6: The simplification algorithm. It asks an oracle to propose a set of possible e', and reports if any of them is equivalent to e with the well-defined program assumption.

Theorem 2 (Simplification)**.** In a well-defined program $\mathcal{P}$, an optimizer can simplify expression e with another e', if there is no input $\mathbf{x}$ that evaluates $e(\mathbf{x})$ and $e'(\mathbf{x})$ to different values, while both reaching e and satisfying the well-defined program assumption $\Delta(\mathbf{x})$:

$$\exists e' \nexists \mathbf{x}\colon e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}). \tag{4}$$

The boolean expression $e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ is referred as the *simplification query*.

Proof. Assuming $\Delta(\mathbf{x})$ is *true*, if the simplification query $e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ always evaluates to *false*, then either $e(\mathbf{x}) = e'(\mathbf{x})$, meaning that they evaluate to the same value; or $R_e(\mathbf{x})$ is *false*, meaning that e is unreachable. In either case, one can safely replace e with e'. □

Simplification relies on an oracle to propose e' for a given expression e. Note that there is no restriction on the proposed expression e'. In practice, it should be simpler than the original e since compilers tend to simplify code. STACK currently implements two oracles:

- Boolean oracle: propose *true* and *false* in turn for a boolean expression, enumerating possible values.
- Algebra oracle: propose to eliminate common terms on both sides of a comparison if one side is a subexpression of the other. It is useful for simplifying non-constant expressions, such as proposing $y < 0$ for $x + y < x$, by eliminating x from both sides.

As an example, consider simplifying $p + 100 < p$ using the boolean oracle, where p is a pointer. For simplicity assume its reachability condition is *true*. From Figure 3, the UB condition of $p + 100$ is $p_\infty + 100_\infty \notin [0, 2^n - 1]$. The boolean oracle first proposes *true*. The corresponding simplification query is:

$$\begin{aligned}&(p + 100 < p) \neq \mathit{true}\\ \wedge\ &\mathit{true} \wedge (\mathit{true} \rightarrow \neg(p_\infty + 100_\infty \notin [0, 2^n - 1])).\end{aligned}$$

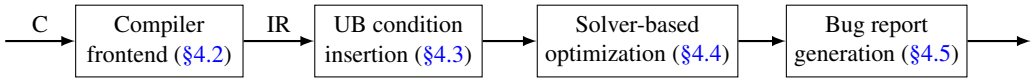

Figure 7: STACK's workflow. It invokes clang to convert a C/C++ program into LLVM IR, and then detects unstable code based on the IR.

Clearly, this is satisfiable. The boolean oracle then proposes *false*. This time the simplification query is:

$$(p + 100 < p) \neq \mathit{false} \\ \wedge \mathit{true} \wedge (\mathit{true} \rightarrow \neg(p_\infty + 100_\infty \notin [0, 2^n - 1])) .$$

Since there is no pointer p that satisfies this query, one can fold $p + 100 < p$ into *false*. §6.2.2 will show more examples of identifying unstable code using simplification.

With the above definition it is straightforward to construct an algorithm to identify unstable code due to simplification (see Figure 6). The algorithm consults an oracle for every possible simpler form e' for expression e. Similarly to elimination, it warns if it finds e' that is equivalent to e only with the well-defined program assumption.

3.3 Discussion

The model focuses on discarding unstable code by exploring two basic optimizations, elimination because of unreachability and simplification because of unnecessary computation. It is possible to exploit the well-defined program assumption in other forms. For example, instead of discarding code, some optimizations reorder instructions and produce unwanted code due to memory aliasing [47] or data races [2], which STACK does not model.

STACK implements two oracles, boolean and algebra, for proposing new expressions for simplification. One can extend it by introducing new oracles.

4 Design

This section describes the design of the STACK checker that detects unstable code by mimicking an aggressive compiler. A challenge in designing STACK is to make it scale to large programs. To address this challenge, STACK uses variants of the algorithms presented in §3 that work on individual functions. A further challenge is to avoid reporting false warnings for unstable code that is generated by the compiler itself, such as macros and inlined functions.

4.1 Overview

STACK works in four stages, as illustrated in Figure 7. In the first stage, a user prepends a script `stack-build` to the actual building command, such as:

```
% stack-build make
```

The script `stack-build` intercepts invocations to gcc and invokes clang instead to compile source code into the LLVM intermediate representation (IR). The remaining three stages work on the IR.

In the second stage, STACK inserts UB conditions listed in Figure 3 into the IR. In the third stage, it performs a solver-based optimization using a variant of the algorithms described in §3.2. In the fourth stage, STACK generates a bug report of unstable code discarded by the solver-based optimization, with the corresponding set of UB conditions. For example, for Figure 2 STACK links the null pointer check `!tun` to the earlier pointer dereference `tun->sk`.

4.2 Compiler frontend

STACK invokes clang to compile C-family source code to the LLVM IR for the rest of the stages. Furthermore, to detect unstable code across functions, it invokes LLVM to inline functions, and works on individual functions afterwards for better scalability.

A challenge is that STACK should focus on unstable code written by programmers, and ignore code generated by the compiler (e.g., from macros and inline functions). Consider the code snippet below:

```
#define IS_A(p) (p != NULL && p->tag == TAG_A)
p->tag == ...;
if (IS_A(p)) ...;
```

Assume `p` is a pointer passed from the caller. Ideally, STACK could inspect the callers and check whether `p` can be null. However, STACK cannot do this because it works on individual functions. STACK would consider the null pointer check `p != NULL` unstable due to the earlier dereference `p->tag`. In our experience, this causes a large number of false warnings, because programmers do not directly write the null pointer check but simply reuse the macro `IS_A`. To reduce false warnings, STACK ignores such compiler-generated code by tracking code origins, at the cost of missing possible bugs (see §4.6).

To do so, STACK implements a clang plugin to record the original macro for macro-expanded code in the IR during preprocessing and compilation. Similarly, it records the original function for inlined code in the IR during inlining. The final stage uses the recorded origin information to avoid generating bug reports for compiler-generated unstable code (see §4.5).

4.3 UB condition insertion

STACK implements the UB conditions listed in Figure 3. For each UB condition, STACK inserts a special function call into the IR at the corresponding instruction:

```
void bug_on(bool expr);
```

This function takes one boolean argument as the UB condition of the instruction.

It is straightforward to represent UB conditions as a boolean argument in the IR. For example, for a division x/y, Stack inserts bug_on($y = 0$) for division by zero. The next stage uses these bug_on calls to compute the well-defined program assumption.

4.4 Solver-based optimization

To detect unstable code, Stack runs the algorithms described in §3.2 in the following order:

- elimination,
- simplification with the boolean oracle, and
- simplification with the algebra oracle.

To implement these algorithms, Stack consults the Boolector solver [3] to decide satisfiability for elimination and simplification queries, as shown in (3) and (4). Both queries need to compute the terms $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$. However, it is practically infeasible to precisely compute them for large programs. By definition, computing the reachability condition $R_e(\mathbf{x})$ requires inspecting all paths from the start of the program, and computing the well-defined program assumption $\Delta(\mathbf{x})$ requires inspecting the entire program for UB conditions. Neither scales to a large program.

To address this challenge, Stack computes approximate queries by limiting the computation to a single function. To describe the impact of this change, we use the following two terms. First, let $R'_e(\mathbf{x})$ denote fragment e's reachability condition from the start of current function; Stack replaces $R_e(\mathbf{x})$ with R'_e. Second, let $\mathrm{dom}(e)$ denote e's dominators [35: §7.3], the set of fragments that every execution path reaching e must have reached; Stack replaces the well-defined program assumption $\Delta(\mathbf{x})$ over the entire program with that over $\mathrm{dom}(e)$.

With these terms we describe the variant of the algorithms for identifying unstable code by computing approximate queries. Stack eliminates fragment e if the following query is unsatisfiable:

$$R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{5}$$

Similarly, Stack simplifies e into e' if the following query is unsatisfiable:

$$e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{6}$$

Appendix A provides a proof that using both approximate queries still correctly identifies unstable code.

Stack computes the approximate queries as follows. To compute the reachability condition $R'_e(\mathbf{x})$ within current function, Stack uses Tu and Padua's algorithm [48]. To compute the UB condition $\bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x})$, Stack collects them from the bug_on calls within e's dominators.

```
1: procedure MinUBCond(Q_e [= H ∧ ⋀_{d∈dom(e)} ¬U_d(x)])
2:     ubset ← ∅
3:     for all d ∈ dom(e) do
4:         Q'_e ← H ∧ ⋀_{d'∈dom(e)\{d}} ¬U_{d'}(x)
5:         if Q'_e is SAT then
6:             ubset ← ubset ∪ {U_d}
7:     return ubset
```

Figure 8: Algorithm for computing the minimal set of UB conditions that lead to unstable code given query Q_e for fragment e.

4.5 Bug report generation

Stack generates a bug report for unstable code based on the solver-based optimization. First, it inspects the recorded origin of each unstable code case in the IR, and ignores code that is generated by the compiler, rather than written by the programmer.

To help users understand the bug report, Stack reports the minimal set of UB conditions that make each report's code unstable [8], using the following greedy algorithm.

Let Q_e be the query with which Stack decided that fragment e is unstable. The query Q_e then must be unsatisfiable. From (5) and (6), we know that the query must be in the following form:

$$Q_e = H \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{7}$$

H denotes the term(s) excluding $\bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x})$ in Q_e. The goal is to find the minimal set of UB conditions that help make Q_e unsatisfiable.

To do so, Stack masks out each UB condition in e's dominators from Q_e individually to form a new query Q'_e; if the new query Q'_e becomes satisfiable, then the UB condition masked out is crucial for making fragment e unstable. The complete algorithm is listed in Figure 8.

4.6 Limitations

The list of undefined behavior Stack implements (see Figure 3) is incomplete. For example, it misses violations of strict aliasing [24: §6.5] and uses of uninitialized variables [24: §6.3.2.1]. We decided not to implement them because gcc already issues decent warnings for both cases. It would be easy to extend Stack to do so as well.

Moreover, since our focus is to find subtle code changes due to optimizations, we choose not to implement undefined behavior that occurs in the frontend. One example is evaluating `(x = 1) + (x = 2)`; this fragment has undefined behavior due to "unsequenced side effects" [24: §6.5/p2]. We believe that the frontend rather than the optimizer should be able to warn against such cases.

As discussed in §4.4, Stack implements approximation algorithms for better scalability, using approximate reachability and UB conditions. Stack may miss unstable code due to these approximations. As Stack consults a constraint solver with elimination and simplification queries,

	# bugs	pointer	null	integer	div	shift	buffer	abs	memcpy	free	realloc
Binutils	8	6	1			1					
e2fsprogs	3		1			1					1
FFmpeg+Libav	21	9	6	1	1	3	1				
FreeType	3	3									
GRUB	2		2								
HiStar [52]	3	1	2								
Kerberos	11	1	9	1							
libX11	2										2
libarchive	2			2							
libgcrypt	2				2						
Linux kernel	32	1	6	1	2	10	5		5	2	
Mozilla	3		2			1					
OpenAFS	11		6				4	1			
plan9port	3	1	1	1							
Postgres	9		1	7			1				
Python	5	5									
QEMU	4					3			1		
Ruby+Rubinius	2		1		1						
Sane	8				1					7	
uClibc	2			2							
VLC	2						2				
Xen	3	1	1			1					
Xpdf	9			8		1					
others (⋆)	10	1	5			2	1		1		
all	160	29	44	23	7	23	14	1	7	9	3

(⋆) Bionic, Dune [1], file, GMP, Mosh [51], MySQL, OpenSSH, OpenSSL, PHP, Wireshark.

Figure 9: New bugs identified by STACK. We also break down the number of bugs by undefined behavior from Figure 3: "pointer" (pointer overflow), "null" (null pointer dereference), "integer" (signed integer overflow), "div" (division by zero), "shift" (oversized shift), "buffer" (buffer overflow), "abs" (absolute value overflow), "memcpy" (overlapped memory copy), "free" (use after free), and "realloc" (use after realloc).

STACK will also miss unstable code if the solver times out. See §6.6 for a completeness evaluation.

STACK reports false warnings when it flags redundant code as unstable, as programmers sometimes simply write useless checks that have no effects (see §6.2.4). Note that even though such redundant code fragments are false warnings, discarding them is allowed by the specification.

5 Implementation

We implemented STACK using the LLVM compiler framework [30] and the Boolector solver [3]. STACK consists of approximately 4,000 lines of C++ code.

6 Evaluation

This section answers the following questions:

- Is STACK useful for finding new bugs? (§6.1)
- What kinds of unstable code does STACK find? (§6.2)
- How precise are STACK's bug reports? (§6.3)
- How long does STACK take to analyze a large system? (§6.4)
- How prevalent is unstable code in real systems, and what undefined behavior causes it? (§6.5)
- What unstable code does STACK miss? (§6.6)

6.1 New bugs

From July 2012 to March 2013, we periodically applied STACK to systems software written in C/C++ to identify unstable code. The systems STACK analyzed are listed in Figure 9, and include OS kernels, virtual machines, databases, multimedia encoders/decoders, language runtimes, and security libraries. Based on STACK's bug reports, we submitted patches to the corresponding developers. The developers confirmed and fixed 160 new bugs. The results show that unstable code is widespread, and that STACK is useful for identifying unstable code.

We also break down the bugs by type of undefined behavior. The results show that several kinds of undefined behavior contribute to the unstable code bugs.

6.2 Analysis of bug reports

This subsection reports our experience of finding and fixing unstable code with the aid of STACK. We manu-

```
int64_t arg1 = ...;
int64_t arg2 = ...;
if (arg2 == 0)
    ereport(ERROR, ...);
int64_t result = arg1 / arg2;
if (arg2 == -1 && arg1 < 0 && result <= 0)
    ereport(ERROR, ...);
```

Figure 10: An invalid signed division overflow check in Postgres, where the division precedes the check. A malicious SQL query will crash it on x86-64 by exploiting signed division overflow.

```
char buf[15]; /* filled with data from user space */
unsigned long node;
char *nodep = strchr(buf, '.') + 1;
if (!nodep)
    return -EIO;
node = simple_strtoul(nodep, NULL, 10);
```

Figure 11: An incorrect null pointer check in the Linux sysctl implementation for `/proc/sys/net/decnet/node_address`. A correct null check should test the result of `strchr`, rather than that plus one, which is always non-null.

ally classify Stack's bug reports into the following four categories based on the impact:

- non-optimization bugs, causing problems regardless of optimizations;
- urgent optimization bugs, where existing compilers are known to cause problems with optimizations turned on, but not with optimizations turned off;
- time bombs, where no known compilers listed in §2.3 cause problems with optimizations, though Stack does and future compilers may do so as well; and
- redundant code: false warnings, such as useless checks that compilers can safely discard.

The rest of this subsection illustrates each category using examples from Stack's bug reports. All the bugs described next were previously unknown but now have been confirmed and fixed by the corresponding developers.

6.2.1 Non-optimization bugs

Non-optimization bugs are unstable code that causes problems even without optimizations, such as the null pointer dereference bug shown in Figure 2, which directly invokes undefined behavior.

To illustrate the subtle consequences of invoking undefined behavior, consider the implementation of the 64-bit signed division operator for SQL in the Postgres database, as shown in Figure 10. The code first rejects the case where the divisor is zero. Since 64-bit integers range from -2^{63} to $2^{63}-1$, the only overflow case is $-2^{63}/-1$, where the expected quotient 2^{63} exceeds the range and triggers undefined behavior. The Postgres developers incorrectly assumed that the quotient must wrap around to -2^{63} in this case, as in some higher-level languages (e.g., Java), and tried to catch it by examining the overflowed quotient *after* the division, using the following check:

```
arg2 == -1 && arg1 < 0 && arg1 / arg2 <= 0.
```

Stack identifies this check as unstable code: the division implies that the overflow must *not* occur to avoid undefined behavior, and thus the overflow check after the division must be *false*.

While signed division overflow is undefined behavior in C, the corresponding x86-64 instruction `IDIV` traps on overflow. One can exploit this to crash the database server on x86-64 by submitting a SQL query that invokes $-2^{63}/-1$, such as:

```
SELECT ((-9223372036854775808)::int8) / (-1);
```

Interestingly, we notice that the Postgres developers tested the $-2^{63}/-1$ crash in 2006, but incorrectly concluded that this "seemed OK" [34]. We believe the reason is that they tested Postgres on x86-32, where there was no 64-bit `IDIV` instruction. In that case, the compiler would generate a call to a library function `lldiv` for 64-bit signed division, which returns -2^{63} for $-2^{63}/-1$ rather than a hardware trap. The developers hence overlooked the crash issue.

To fix this bug, we submitted a straightforward patch that checks whether `arg1` is -2^{63} and `arg2` is -1 before `arg1/arg2`. However, the Postgres developers insisted on their own fix. Particularly, instead of directly comparing `arg1` with -2^{63}, they chose the following check:

```
arg1 != 0 && (-arg1 < 0) == (arg1 < 0).
```

Stack identifies this check as unstable code for similar reasons: the negation `-arg1` implies that `arg1` cannot be -2^{63} to avoid undefined behavior, and thus the check must be *false*. We will further analyze this check in §6.2.3.

By identifying unstable code, Stack is also useful for uncovering programming errors that do not directly invoke undefined behavior. Figure 11 shows an incorrect null pointer check from the Linux kernel. The intention of this check was to reject a network address without any dots. Since `strchr(buf, '.')` returns null if it cannot find any dots in `buf`, a correct check should check whether its result is null, rather than that plus one. One can bypass the check `!nodep` with a malformed network address from user space and trigger an invalid read at page zero. Stack identifies the check `!nodep` as unstable code, because under the no-pointer-overflow assumption `nodep` (a pointer plus one) must be non-null.

6.2.2 Urgent optimization bugs

Urgent optimization bugs are unstable code that existing compilers already optimize to cause problems. §2.2 described a set of examples where compilers either discard the unstable code or rewrite it into some vulnerable form.

To illustrate the consequences, consider the code snippet from FFmpeg/Libav for parsing Adobe's Action Message Format, shown in Figure 12. The parsing code starts

```
const uint8_t *data     = /* buffer head */;
const uint8_t *data_end = /* buffer tail */;
int size = bytestream_get_be16(&data);
if (data + size >= data_end || data + size < data)
    return -1;
data += size;
...
int len = ff_amf_tag_size(data, data_end);
if (len < 0 || data + len >= data_end
            || data + len < data)
    return -1;
data += len;
/* continue to read data */
```

Figure 12: Unstable bounds checks in the form `data + x < data` from FFmpeg/Libav, which gcc optimizes into $x < 0$.

```
void pdec(io *f, int k) {
    if (k < 0) {          /* print negative k */
        if (-k >= 0) {    /* not INT_MIN?     */
            pchr(f, '-'); /* print minus      */
            pdec(f, -k);  /* print -k         */
            return;
        }
        ...               /* print INT_MIN    */
        return;
    }
    ...                   /* print positive k */
}
```

Figure 13: An unstable integer check in plan9port. The function `pdec` prints a signed integer `k`; gcc optimizes the check `-k >= 0` into *true* when it learns that `k` is negative, leading to an infinite loop if the input `k` is `INT_MIN`.

with two pointers, `data` pointing to the head of the input buffer, and `data_end` pointing to one past the end. It first reads in an integer `size` from the input buffer, and fails if the pointer `data + size` falls out of the bounds of the input buffer (i.e., between `data` and `data_end`). The intent of the check `data + size < data` is to reject a large `size` that causes `data + size` to wrap around to a smaller pointer and bypass the earlier check `data + size >= data_end`. The parsing code later reads in another integer `len` and performs similar checks.

Stack identifies the two pointer overflow checks in the form `data + x < data` as unstable code, where x is a signed integer (e.g., `size` and `len`). Specifically, with the algebra oracle Stack simplifies the check `data + x < data` into $x < 0$, and warns against this change. Note that this is slightly different from Figure 1: x is a signed integer, rather than unsigned, so the check is not always *false* under the well-defined program assumption.

Both gcc and clang perform similar optimizations, by rewriting `data + x < data` into $x < 0$. As a result, a large `size` or `len` from malicious input is able to bypass the checks, leading to an out-of-bounds read. A correct fix is to replace `data + x >= data_end || data + x < data` with `x >= data_end - data`, which is simpler and also avoids invoking undefined behavior; one should also add the check $x < 0$ if x can be negative.

```
int64_t arg1 = ...;
if (arg1 != 0 && ((-arg1 < 0) == (arg1 < 0)))
    ereport(ERROR, ...);
```

Figure 14: A time bomb in Postgres. The intention is to check whether `arg1` is the most negative value -2^{n-1}, similar to Figure 13.

```
struct p9_client *c = ...;
struct p9_trans_rdma *rdma = c->trans;
...
if (c)
    c->status = Disconnected;
```

Figure 15: Redundant code from the Linux kernel, where the caller of this code snippet ensures that `c` must be non-null and the null pointer check against `c` is always *true*.

Figure 13 shows an urgent optimization bug that leads to an infinite loop from plan9port. The function `pdec` is used to print a signed integer k; if k is negative, the code prints the minus symbol and then invokes `pdec` again with the negation $-k$. Assuming k is an n-bit integer, one special case is k being -2^{n-1} (i.e., `INT_MIN`), the negation of which is undefined. The programmers incorrectly assumed that `-INT_MIN` would wrap around to `INT_MIN` and remain negative, so they used the check $-k >= 0$ to filter out `INT_MIN` when k is known to be negative.

Stack identifies the check $-k >= 0$ as unstable code; gcc also optimizes the check into *true* as it learns that k is negative from the earlier $k < 0$. Consequently, invoking `pdec` with `INT_MIN` will lead an infinite loop, printing the minus symbol repeatedly. A simple fix is to replace $-k >= 0$ with a safe form k `!= INT_MIN`.

6.2.3 Time bombs

A time bomb is unstable code that is harmless at present, since no compiler listed in §2.3 can currently optimize it. But this situation may change over time. §2.3 already showed how past compiler changes trigger time bombs to become urgent optimization bugs. §6.2.1 illustrated how a time bomb in Postgres emerged as the x86 processor evolved: the behavior of 64-bit signed division on overflow changed from silent wraparound to trap, allowing one to crash the database server with malicious SQL queries.

Figure 14 shows a time bomb example from Postgres. As mentioned in §6.2.1, the Postgres developers chose this approach to check whether `arg1` is -2^{63} without using the constant value of -2^{63}; their assumption was that the negation of a non-zero integer would have a different sign unless it is -2^{63}.

The code currently works; the time bomb does not go off, and does not cause any problems, unlike its "equivalent" form in Figure 13. This luck relies on the fact that no production compilers discard it. Nonetheless, Stack identifies the check as unstable code, and we believe that some research compilers such as Bitwise [43] already discard the check. Relying on compilers to not optimize time

	build time	analysis time	# files	# queries	# query timeouts
Kerberos	1 min	2 min	705	79,547	2 (0.003%)
Postgres	1 min	11 min	770	229,624	1,131 (0.493%)
Linux kernel	33 min	62 min	14,136	3,094,340	1,212 (0.039%)

Figure 16: STACK's performance numbers when running it against Kerberos, Postgres, and the Linux kernel, including the build time, the analysis time, the number of files, the number of total queries STACK made, and the number of queries that timed out.

bombs for system security is risky, and we recommend fixing problems flagged by STACK to avoid this risk.

6.2.4 Redundant code

Figure 15 shows an example of redundant code from the Linux kernel. STACK identifies the null pointer check against the pointer `c` in the `if` condition as unstable code, due to the earlier dereference `c->trans`. The caller of the code snippet ensures that the pointer `c` must be non-null, so the check is always *true*. Our experience shows that redundant code comprises only a small portion of unstable code that STACK reports (see §6.3).

Depending on their coding conventions, it is up to programmers to decide whether to keep redundant code. Based on the feedback from STACK's users, we have learned that programmers often prefer to remove such redundant checks or convert them to assertions for better code quality, even if they are not real bugs.

6.3 Precision

To understand the precision of STACK's results, we further analyzed every bug report STACK produced for Kerberos and Postgres. The results below show that STACK has a low rate of false warnings (i.e., redundant code).

Kerberos. STACK reported 11 bugs in total, all of which were confirmed and fixed by the developers. In addition, the developers determined that one of them was remotely exploitable and requested a CVE identifier (CVE-2013-1415) for this bug. After the developers fixed these bugs, STACK produced zero reports.

Postgres. STACK reported 68 bugs in total. The developers promptly fixed 9 of them after we demonstrated how to crash the database server by exploiting these bugs, as described in §6.2.1. We further discovered that Intel's icc and PathScale's pathcc compilers discarded 29 checks, which STACK identified as unstable code (i.e., urgent optimization bugs), and reported these problems to the developers. At the writing of this paper, the strategies for fixing them are still under discussion.

STACK found 26 time bombs (see §6.2.3 for one example); we did not submit patches to fix these time bombs given the developers' hesitation in fixing urgent optimization bugs. STACK also produced 4 bug reports that identified redundant code, which did not need fixing.

algorithm	# reports	# packages
elimination	23,969	2,079
simplification (boolean oracle)	47,040	2,672
simplification (algebra oracle)	871	294

Figure 17: Number of reports generated by each of STACK's algorithms from §3.2 for all Debian Wheezy packages, and the number of packages for which at least one such report was generated.

6.4 Performance

To measure the running time of STACK, we ran it against Kerberos, Postgres, and the Linux kernel (with all modules enabled), using their source code from March 23, 2013. The experiments were conducted on a 64-bit Ubuntu Linux machine with an Intel Core i7-980 3.3 GHz CPU and 24 GB of memory. The processor has 6 cores, and each core has 2 hardware threads.

STACK built and analyzed each package using 12 processes in parallel. We set a timeout of 5 seconds for each query to the solver (including computing the UB condition set as described in §4.5). Figure 16 lists the build time, the analysis time, the number of files, the number of total queries to the solver, and the number of query timeouts. The results show that STACK can finish analyzing a large system within a reasonable amount of time.

We noticed a small number of solver timeouts (less than 0.5%) due to complex reachability conditions, often at the end of a function. STACK would miss unstable code in such cases. To avoid this, one can increase the timeout.

6.5 Prevalence of unstable code

We applied STACK to all 17,432 packages in the Debian Wheezy archive as of March 24, 2013. STACK checked 8,575 of them that contained C/C++ code. Building and analyzing these packages took approximately 150 CPU-days on Intel Xeon E7-8870 2.4 GHz processors.

For 3,471 out of these 8,575 packages, STACK detected at least one instance of unstable code. This suggests that unstable code is a widespread problem.

Figure 17 shows the number of reports generated by each of STACK's algorithms. These results suggest that they are all useful for identifying unstable code.

Each of STACK's reports contains a set of UB conditions that cause the code to be unstable. Figure 18 shows the number of times each kind of UB condition showed up in a report. These numbers confirm that many kinds of undefined behavior lead to unstable code in practice.

UB condition	# reports	# packages
null pointer dereference	59,230	2,800
buffer overflow	5,795	1,064
signed integer overflow	4,364	780
pointer overflow	3,680	614
oversized shift	594	193
aliasing	330	70
overlapping memory copy	227	47
division by zero	226	95
use after free	156	79
other libc (cttz, ctlz)	132	7
absolute value overflow	86	23
use after realloc	22	10

Figure 18: Number of reports that involve each of STACK's UB conditions from Figure 3 for all Debian Wheezy packages, and the number of packages for which at least one such report was generated.

As described in §4.5, STACK computes a minimal set of UB conditions necessary for each instance of unstable code. Most unstable code reports (69,301) were the result of just one UB condition, but there were also 2,579 reports with more than one UB condition, and there were even 4 reports involving eight UB conditions. These numbers confirm that some unstable code is caused by multiple undefined behaviors, which suggests that automatic tools such as STACK are necessary to identify them. Programmers are unlikely to find them by manual inspection.

6.6 Completeness

STACK is able to identify all the unstable code examples described in §2.3. However, it is difficult to know precisely how much unstable code STACK would miss in general. Instead we analyze what kind of unstable code STACK misses. To do so, we collected all examples from Regehr's "undefined behavior consequences contest" winners [40] and Wang et al.'s undefined behavior survey [49] as a benchmark, a total of ten tests from real systems.

STACK identified unstable code in seven out of the ten tests. STACK missed three for the following reasons. As described in §4.6, STACK missed two because we chose not to implement their UB conditions for violations of strict aliasing and uses of uninitialized variables; it would be easy to extend STACK to do so. The other case STACK missed was due to approximate reachability conditions, also mentioned in §4.6.

7 Related work

To the best of our knowledge, we present the first definition and static checker to find unstable code, but we build on several pieces of related work. In particular, earlier surveys [26, 41, 49] and blog posts [29, 39, 40] collect examples of unstable code, which motivated us to tackle this problem. We were also motivated by related techniques that can help with addressing unstable code, which we discuss next.

Testing strategies. Our experience with unstable code shows that in practice it is difficult for programmers to notice certain critical code fragments disappearing from the running system as they are silently discarded by the compiler. Maintaining a comprehensive test suite may help catch "vanished" code in such cases, though doing so often requires a substantial effort to achieve high code coverage through manual test cases. Programmers may also need to prepare a variety of testing environments as unstable code can be hardware- and compiler-dependent.

Automated tools such as KLEE [4] can generate test cases with high coverage using symbolic execution. These tools, however, often fail to model undefined behavior correctly. Thus, they may interpret the program differently from the language standard and miss bugs. Consider a check $x + 100 < x$, where x is a signed integer. KLEE considers $x + 100$ to wrap around given a large x; in other words, the check catches a large x when executing in KLEE, even though gcc discards the check. Therefore, to detect unstable code, these tools need to be augmented with a model of undefined behavior, such as the one we proposed in this paper.

Optimization strategies. We believe that programmers should avoid undefined behavior, and we provide suggestions for fixing unstable code in §6.2. However, overly aggressive compiler optimizations are also responsible for triggering these bugs. Traditionally, compilers focused on producing fast and small code, even at the price of sacrificing security, as shown in §2.2. Compiler writers should rethink optimization strategies for generating secure code.

Consider $x + 100 < x$ with a signed integer x again. The language standard does allow compilers to consider the check to be *false* and discard it. In our experience, however, it is unlikely that the programmer intended the code to be removed. A programmer-friendly compiler could instead generate efficient overflow checking code, for example, by exploiting the overflow flag available on many processors after evaluating $x + 100$. This strategy, also allowed by the language standard, produces more secure code than discarding the check. Alternatively, the compiler could produce warnings when exploiting undefined behavior in a potentially surprising way [19].

Currently, gcc provides several options to alter the compiler's assumptions about undefined behavior, such as

- `-fwrapv`, assuming signed integer wraparound for addition, subtraction, and multiplication;
- `-fno-strict-overflow`, assuming pointer arithmetic wraparound in addition to `-fwrapv`; and
- `-fno-delete-null-pointer-checks` [44], assuming unsafe null pointer dereferences.

These options can help reduce surprising optimizations, at the price of generating slower code. However, they cover an incomplete set of undefined behavior that may cause unstable code (e.g., no options for shift or division). Another downside is that these options are specific to gcc; other compilers may not support them or interpret them in a different way [49].

Checkers. Many existing tools can detect undefined behavior as listed in Figure 3. For example, gcc provides the `-ftrapv` option to insert run-time checks for signed integer overflows [42: §3.18]; IOC [11] (now part of clang's sanitizers [9]) and KINT [50] cover a more complete set of integer errors; Saturn [12] finds null pointer dereferences; several dedicated C interpreters such as kcc [14] and Frama-C [5] perform checks for undefined behavior. See Chen et al.'s survey [6] for a summary.

In complement to these checkers that directly target undefined behavior, STACK finds unstable code that becomes dead due to undefined behavior. In this sense, STACK can be considered as a generalization of Engler et al.'s inconsistency cross-checking framework [12, 16]. STACK, however, supports more expressive assumptions, such as pointer and integer operations.

Language design. Language designers may reconsider whether it is necessary to declare certain constructs as undefined behavior, since reducing undefined behavior in the specification is likely to avoid unstable code. One example is left-shifting a signed 32-bit one by 31 bits. This is undefined behavior [24: §6.5.7], even though the result is consistently `0x80000000` on most modern processors. The committee for the C++ language standard is already considering this change [33].

8 Conclusion

This paper presented the first systematic study of unstable code, an emerging class of system defects that manifest themselves when compilers discard code due to undefined behavior. Our experience shows that unstable code is subtle and often misunderstood by system programmers, that unstable code prevails in systems software, and that many popular compilers already perform unexpected optimizations, leading to misbehaving or vulnerable systems. We introduced a new model for reasoning about unstable code, and developed a static checker, STACK, to help system programmers identify unstable code. We hope that compiler writers will also rethink optimization strategies against unstable code. Finally, we hope this paper encourages language designers to be careful with using undefined behavior in the language specification. All STACK source code is publicly available at http://css.csail.mit.edu/stack/.

Acknowledgments

We thank Haogang Chen, Victor Costan, Li Lu, John Regehr, Jesse Ruderman, Tom Woodfin, the anonymous reviewers, and our shepherd, Shan Lu, for their help and feedback. This research was supported by the DARPA Clean-slate design of Resilient, Adaptive, Secure Hosts (CRASH) program under contract #N66001-10-2-4089, and by NSF award CNS-1053143.

A Correctness of approximation

As discussed in §3.2, STACK performs an optimization if the corresponding query Q is unsatisfiable. Using an approximate query Q' yields a correct optimization if Q' is weaker than Q (i.e., $Q \rightarrow Q'$): if Q' is unsatisfiable, which enables the optimization, the original query Q must also be unsatisfiable.

To prove the correctness of approximation, it suffices to show that the approximate elimination query (5) is weaker than the original query (3); the simplification queries (6) and (4) are similar. Formally, given code fragment e, it suffices to show the following:

$$R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}) \rightarrow R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{8}$$

Proof. Since e's dominators are a subset of the program, the well-defined program assumption over $\mathrm{dom}(e)$ must be weaker than $\Delta(\mathbf{x})$ over the entire program:

$$\Delta(\mathbf{x}) \rightarrow \bigwedge_{d \in \mathrm{dom}(e)} (R_d(\mathbf{x}) \rightarrow \neg U_d(\mathbf{x})). \tag{9}$$

From the definition of $\mathrm{dom}(e)$, if fragment e is reachable, then its dominators must be reachable as well:

$$\forall d \in \mathrm{dom}(e) \colon R_e(\mathbf{x}) \rightarrow R_d(\mathbf{x}). \tag{10}$$

Combining (9) and (10) gives:

$$\Delta(\mathbf{x}) \rightarrow (R_e(\mathbf{x}) \rightarrow \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x})). \tag{11}$$

With $R_e(\mathbf{x})$, we have:

$$R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}) \rightarrow R_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{12}$$

By definition $R_e(\mathbf{x}) \rightarrow R'_e(\mathbf{x})$, so (12) implies (8). □

References

[1] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–348, Hollywood, CA, Oct. 2012.

[2] H.-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 261–268, Chicago, IL, June 2005.

[3] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177, York, UK, Mar. 2009.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, Dec. 2008.

[5] G. Canet, P. Cuoq, and B. Monate. A value analysis for C programs. In *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 123–124, Edmonton, Canada, Sept. 2009.

[6] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011.

[7] Chromium. Issue 12079010: Avoid undefined behavior when checking for pointer wraparound, 2013. https://codereview.chromium.org/12079010/.

[8] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *Journal of Artificial Intelligence Research*, 40:701–728, 2011.

[9] Clang. *Clang Compiler User's Manual: Controlling Code Generation*, 2013. http://clang.llvm.org/docs/UsersManual.html#controlling-code-generation.

[10] J. Corbet. Fun with NULL pointers, part 1, July 2009. http://lwn.net/Articles/342330/.

[11] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 760–770, Zurich, Switzerland, June 2012.

[12] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 435–445, San Diego, CA, June 2007.

[13] C. R. Dougherty and R. C. Seacord. C compilers may silently discard some wraparound checks. Vulnerability Note VU#162289, US-CERT, 2008. http://www.kb.cert.org/vuls/id/162289.

[14] C. Ellison and G. Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL)*, pages 533–544, Philadelphia, PA, Jan. 2012.

[15] C. Ellison and G. Roşu. Defining the undefinedness of C. Technical report, University of Illinois, Apr. 2012. http://hdl.handle.net/2142/30780.

[16] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 57–72, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[17] GCC. Bug 30475 - `assert(int+100 > int)` optimized away, 2007. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475.

[18] GCC. Bug 49820 - explicit check for integer negative after abs optimized away, 2011. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=49820.

[19] GCC. Bug 53265 - warn when undefined behavior implies smaller iteration count, 2013. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=53265.

[20] D. Gohman. The nsw story, Nov. 2011. http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-November/045730.html.

[21] IBM. *Power ISA Version 2.06 Revision B, Book I: Power ISA User Instruction Set Architecture*, July 2010.

[22] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A–Z*, Jan. 2013.

[23] ISO/IEC JTC1/SC22/WG14. *Rationale for International Standard - Programming Languages - C*, Apr. 2003.

[24] ISO/IEC JTC1/SC22/WG14. *ISO/IEC 9899:2011, Programming languages - C*, Dec. 2011.

[25] B. Jack. Vector rewrite attack: Exploitable NULL pointer vulnerabilities on ARM and XScale architectures. White paper, Juniper Networks, May 2007.

[26] R. Krebbers and F. Wiedijk. Subtleties of the ANSI/ISO C standard. Document N1639, ISO/IEC JTC1/SC22/WG14, Sept. 2012.

[27] T. Lane. Anyone for adding `-fwrapv` to our standard CFLAGS?, Dec. 2005. http://www.postgresql.org/message-id/1689.1134422394@sss.pgh.pa.us.

[28] T. Lane. Re: gcc versus division-by-zero traps, Sept. 2009. http://www.postgresql.org/message-id/19979.1251998812@sss.pgh.pa.us.

[29] C. Lattner. What every C programmer should know about undefined behavior, May 2011. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.

[30] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, Palo Alto, CA, Mar. 2004.

[31] Linux kernel. Bug 14287 - ext4: fixpoint divide exception at `ext4_fill_super`, 2009. https://bugzilla.kernel.org/show_bug.cgi?id=14287.

[32] D. MacKenzie, B. Elliston, and A. Demaille. *Autoconf: Creating Automatic Configuration Scripts for version 2.69*. Free Software Foundation, Apr. 2012.

[33] W. M. Miller. C++ standard core language defect reports and accepted issues, issue 1457: Undefined behavior in left-shift, Feb. 2012. http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#1457.

[34] B. Momjian. Re: Fix for Win32 division involving `INT_MIN`, June 2006. http://www.postgresql.org/message-id/200606090240.k592eUj23952@candle.pha.pa.us.

[35] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[36] D. Novillo. A propagation engine for GCC. In *Proceedings of the 2005 GCC & GNU Toolchain Developers' Summit*, pages 175–184, Ottawa, Canada, June 2005.

[37] Python. Issue 17016: _sre: avoid relying on pointer overflow, 2013. http://bugs.python.org/issue17016.

[38] S. Ranise, C. Tinelli, and C. Barrett. QF_BV logic, 2013. http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2.

[39] J. Regehr. A guide to undefined behavior in C and C++, July 2010. http://blog.regehr.org/archives/213.

[40] J. Regehr. Undefined behavior consequences contest winners, July 2012. http://blog.regehr.org/archives/767.

[41] R. C. Seacord. Dangerous optimizations and the loss of causality, Feb. 2010. https://www.securecoding.cert.org/confluence/download/attachments/40402999/Dangerous+Optimizations.pdf.

[42] R. M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection for GCC 4.8.0*. Free Software Foundation, 2013.

[43] M. Stephenson, J. Babb, and S. Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 108–120, Vancouver, Canada, June 2000.

[44] E. Teo. [PATCH] add `-fno-delete-null-pointer-checks` to gcc `CFLAGS`, July 2009. https://lists.ubuntu.com/archives/kernel-team/2009-July/006609.html.

[45] J. Tinnes. Bypassing Linux NULL pointer dereference exploit prevention (`mmap_min_addr`), June 2009. http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html.

[46] L. Torvalds. Re: [patch] CFS scheduler, -v8, May 2007. https://lkml.org/lkml/2007/5/7/213.

[47] J. Tourrilhes. Invalid compilation without `-fno-strict-aliasing`, Feb. 2003. https://lkml.org/lkml/2003/2/25/270.

[48] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.

[49] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems*, Seoul, South Korea, July 2012.

[50] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving integer security for systems with Kint. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 163–177, Hollywood, CA, Oct. 2012.

[51] K. Winstein and H. Balakrishnan. Mosh: An interactive remote shell for mobile clients. In *Proceedings of the 2012 USENIX Annual Technical Conference*, pages 177–182, Boston, MA, June 2012.

[52] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 263–278, Seattle, WA, Nov. 2006.