DOI:10.1145/2885256

A Differential Approach to Undefined Behavior Detection

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

Abstract

This paper studies undefined behavior arising in systems programming languages such as C/C++. Undefined behavior bugs lead to unpredictable and subtle systems behavior, and their effects can be further amplified by compiler optimizations. Undefined behavior bugs are present in many systems, including the Linux kernel and the Postgres database. The consequences range from incorrect functionality to missing security checks.

This paper proposes a formal and practical approach, which finds undefined behavior bugs by finding "unstable code" in terms of optimizations that leverage undefined behavior. Using this approach, we introduce a new static checker called Stack that precisely identifies undefined behavior bugs. Applying Stack to widely used systems has uncovered 161 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* (Section 2.3 in Ref.[18]). For instance, in C "use of a nonportable or erroneous program construct or of erroneous data" leads to undefined behavior (Section 3.4.3 in Ref.[23]); a comprehensive list of undefined behavior is available in the C language specification (Section J.2 in Ref.[23]).

One category of undefined behavior is simply programming mistakes, such as buffer overflow and null pointer dereference. The other category is nonportable operations, the hardware implementations of which often have subtle differences. For example, when signed integer overflow or division by zero occurs, a division instruction traps on x86 (Section 3.2 in Ref.[22]), while it silently produces an undefined result on PowerPC (Section 3.3.8 in Ref.[30]). Another example is shift instructions: left-shifting a 32-bit one by 32 bits produces zero on ARM and PowerPC, but one on x86; however, left-shifting a 32-bit one by 64 bits produces zero on ARM, but one on x86 and PowerPC.

By designating certain programming mistakes and nonportable operations as having undefined behavior, the specifications give compilers the freedom to generate instructions that behave in arbitrary ways in those cases, allowing compilers to generate efficient and portable code without extra checks. For example, many higher-level programming languages (e.g., Java) have well-defined handling (e.g., runtime exceptions) on buffer overflow, and the compiler would need to insert extra bounds checks for memory access operations. However, the C/C++ compiler does *not* to need to insert bounds checks, as out-of-bounds cases are undefined. It is the programmer's responsibility to avoid undefined behavior.

According to the C/C++ specifications, programs that invoke undefined behavior can have arbitrary problems. As one summarized, "permissible undefined behavior ranges from ignoring the situation completely with unpredictable results, to having demons fly out of your nose."[45] But what happens in practice? The rest of this paper will show that modern compilers increasingly exploit undefined behavior to perform aggressive optimizations; with these optimizations many programs can produce surprising results that programmers did not anticipate.

2. RISKS OF UNDEFINED BEHAVIOR

One risk of undefined behavior is that a program will observe different behavior on different hardware architectures, operating systems, or compilers. For example, a program that performs an oversized left-shift will observe different results on ARM and x86 processors. As another example, consider a simple SQL query:

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

This query caused signed integer overflow in the Postgres database server, which on a 32-bit Windows system did not cause any problems, but on a 64-bit Windows system caused the server to crash, due to the different behavior of division instructions on the two systems.[44]

In addition, compiler optimizations can amplify the effects of undefined behavior. For example, 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, the Linux kernel, and the Python interpreter.[44]

While this check appears to work with a flat address space, it fails on a segmented architecture (Section 6.3.2.3 in Ref.[32]). Therefore, the C standard states that an overflowed pointer is undefined (Section 6.5.6 in Ref.[23(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

The original version of this paper is entitled "Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior" and was published in the *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.[44]

false. Consequently, gcc removes the check, paving the way for an attack to the system.[17]

As another example, Figure 2 shows a mild defect in the Linux kernel, where the programmer incorrectly placed the dereference `tun->sk` 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[24, 38]), 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, 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 (Section 6.5.3 in Ref.[23]). 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.[13]

Figure 1. A pointer overflow check found in several code bases. The code becomes vulnerable as gcc optimizes away the second `if` statement.[17]

```
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 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.[13]

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

To further understand how compiler optimizations exploit undefined behavior, we conduct a study using six real-world examples in the form of sanity checks, as shown in the top row of Figure 3. All of these checks may evaluate to *false* and become dead code under optimizations, because they invoke undefined behavior. We will use them to test existing compilers next.

- The check $p + 100 < p$ resembles Figure 1.
- The null pointer check $!p$ with an earlier dereference resembles Figure 2.
- The check $x + 100 < x$ with a signed integer x caused a harsh debate in gcc's bugzilla.[5]
- The check $x^+ + 100 < 0$ tests whether optimizations perform more elaborate reasoning; x^+ is known to be positive.
- The shift check $!(1 << x)$ was intended to catch a large shifting amount x, from a patch to the ext4 file system.[6]
- The check `abs`$(x) < 0$, intended to catch the most negative value (i.e., -2^{n-1}), tests whether optimizations understand library functions.[7]

We chose 12 well-known C/C++ compilers to see what they do with the above code examples: 2 open-source compilers (gcc and clang) and 10 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 code example, we test whether a compiler optimizes the check into *false*, and if so, we find the lowest optimization level `-O`*n* at which it happens. The result is shown in Figure 3.

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:

Figure 3. Optimizations of unstable code in popular compilers. This includes 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.

	`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-4.9.1	O2	O2	O2	O2	–	O2
clang-1.0	O1	–	–	–	–	–
clang-3.4	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	–	–	–	–	–

- 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 4.9.1, released in 2014.

For comparison, we chose two versions of clang, 1.0 released in 2009, and 3.4 released in 2014.

We can see that exploiting undefined behavior to eliminate code is common among compilers, not just in recent gcc versions as some programmers have claimed.[26] Even gcc 2.95.3 eliminates $x + 100 < x$. Some compilers eliminates code that gcc does not (e.g., clang on $1 << x$).

These optimizations can lead to baffling results even for veteran C programmers, because code unrelated to the undefined behavior gets optimized away or transformed in unexpected ways. Such bugs lead to spirited debates between compiler developers and practitioners that use the C language but do not adhere to the letter of the official C specification. Practitioners describe these optimizations as "make no sense"[40] and merely the compiler's "creative reinterpretation of basic C semantics."[26] On the other hand, compiler writers argue that the optimizations are legal under the specification; it is the "broken code"[5] that programmers should fix. Worse yet, as compilers evolve, new optimizations are introduced that may break code that used to work before; as we show in Figure 3, many compilers have become more aggressive over the past 20 years with such optimizations.

3. CHALLENGES OF UNDEFINED BEHAVIOR DETECTION

Given the wide range of problems that undefined behavior can cause, what should programmers do about it? The naïve approach is to require programmers to carefully read and understand the C language specification, so that they can write careful code that avoids invoking undefined behavior. Unfortunately, as we demonstrate in Section 2, even experienced C programmers do not fully understand the intricacies of the C language, and it is exceedingly difficult to avoid invoking undefined behavior in practice.

Since optimizations often amplify the problems due to undefined behavior, some programmers (such as the Postgres developers) have tried reducing the compiler's optimization level, so that aggressive optimizations do not take advantage of undefined behavior bugs in their code. As we see in Figure 3, compilers are inconsistent about the optimization levels at which they take advantage of undefined behavior, and several compilers make undefined behavior optimizations even at optimization level zero (which should, in principle, disable all optimizations).

Runtime checks can be used to detect certain undefined behaviors at runtime; for example, gcc provides an `-ftrapv` option to trap on signed integer overflow, and clang provides an `-fsanitize=undefined` option to trap several more undefined behaviors. There have also been attempts at providing a more "programmer-friendly" refinement of C,[14, 29] which has less undefined behavior, though in general it remains unclear how to outlaw undefined behavior from the specification without incurring significant performance overhead.[14, 42]

Certain static-analysis and model checkers identify classes of bugs due to undefined behavior. For example, compilers can catch some obvious cases (e.g., using gcc's `-Wall`), but in general this is challenging (Part 3 in Ref.[27]); tools that find buffer overflow bugs[11] can be viewed as finding undefined behavior bugs, because referencing a location outside of a buffer's range is undefined behavior. See Section 6 for a more detailed discussion of related work.

4. APPROACH: FINDING DIVERGENT BEHAVIOR

Ideally, compilers would generate warnings for developers when an application invokes undefined behavior, and this paper takes a static analysis approach to finding undefined behavior bugs. This boils down to deciding, for each operation in the program, whether it can be invoked with arguments that lead to undefined behavior. Since many operations in C can invoke undefined behavior (e.g., signed integer operations, pointer arithmetic), producing a warning for every operation would overwhelm the developer, so it is important for the analysis to be precise. Global reasoning can precisely determine what values an argument to each operation can take, but it does not scale to large programs.

Instead of performing global reasoning, our goal is to find local invariants (or likely invariants) on arguments to a given operation. We are willing to be incomplete: if there are not enough local invariants, we are willing to not report potential problems. On the other hand, we would like to ensure that every report is likely to be a real problem.[1]

The local likely invariant that we exploit in this paper has to do with unnecessary source code written by programmers. By "unnecessary source code" we mean dead code, unnecessarily complex expressions that can be transformed into a simpler form, etc. We expect that all of the source code that programmers write should either be necessary code, or it should be clearly unnecessary; that is, it should be clear from local context that the code is unnecessary, without relying on subtle semantics of the C language. For example, the programmer might write `if (0) {...}`, which is clearly unnecessary code. However, our likely invariant tells us that programmers would never write code like `a = b << c; if (c >= 32) {...}`, where `b` is a 32-bit integer. The `if` statement in this code snippet is unnecessary code, because `c` could never be 32 or greater due to undefined behavior in the preceding left-shift. The core of our invariant is that programmers are unlikely to write such subtly unnecessary code.

To formalize this invariant, we need to distinguish "live code" (code that is always necessary), "dead code" (code that is always unnecessary), and "unstable code" (code that is subtly unnecessary). We do this by considering the different possible interpretations that the programmer might have for the C language specification. In particular, we consider C to be the language's official specification, and C′ to be the specification that the programmer believes C has. For the purposes of this paper, C′ differs from C in which operations lead to undefined behavior. For example, a programmer might expect shifts to be well-defined for all possible arguments; this is one such possible C′. In other words, C′ is a relaxed version of the official C, by assigning certain interpretations to operations that have undefined behavior in C.

Using the notion of different language specifications, we say that a piece of code is *live* if, for every possible C′, the code is necessary. Conversely, a piece of code is *dead* if, for every possible C′, the code is unnecessary; this captures code like `if (0) {...}`. Finally, a piece of code is *unstable* if, for some C′ variants, it is unnecessary, but in other C′ variants, it is necessary. This means that two programmers that do not precisely understand the details of the C specification might disagree about what the code is doing. As we demonstrate in the rest of this paper, this heuristic often indicates the presence of a bug.

Building on this invariant, we can now detect when a program is likely invoking undefined behavior. In particular, given an operation *o* in a function *f*, we compute the set of unnecessary code in *f* under different interpretations of undefined behavior at *o*. If the set of unnecessary code is the same for all possible interpretations, we cannot say anything about whether *o* is likely to invoke undefined behavior. However, if the set of unnecessary code varies depending on what undefined behavior *o* triggers, this means that the programmer wrote unstable code. However, by our assumption, this should never happen, and we conclude that the programmer was likely thinking they're writing live code, and simply did not realize that *o* would trigger undefined behavior for the *same* set of inputs that are required for the code to be live.

5. THE STACK TOOL

To find undefined behavior bugs using the above approach, we built a static analysis tool called STACK. In practice, it is difficult to enumerate and consider all possible C′ variants. Thus, to build a practical tool, we pick a single variant, called $C^\star$. $C^\star$ defines a null pointer that maps to address zero, and wrap-around semantics for pointer and integer arithmetic.[31] We believe this captures the common semantics that programmers (mistakenly) believe C provides. Although our $C^\star$ deals with only a subset of undefined behaviors in the C specification, a different $C^\star$ could capture other semantics that programmers might implicitly assume, or handle undefined behavior for other operations that our $C^\star$ does not address.

STACK relies on an optimizer $\mathcal{O}$ to implicitly flag unnecessary code. STACK's $\mathcal{O}$ eliminates dead code and performs expression simplifications under the semantics of C and $C^\star$, respectively. For code fragment *e*, if $\mathcal{O}$ is *not* able to rewrite *e* under neither semantics, STACK considers *e* as "live code"; if $\mathcal{O}$ is able to rewrite *e* under both semantics, *e* is "dead code"; if $\mathcal{O}$ is able to rewrite *e* under C but not $C^\star$, STACK reports it as "unstable code."

Since STACK uses just two interpretations of the language specification (namely, C and $C^\star$), it might miss bugs that could arise under different interpretations. For instance, any code eliminated by $\mathcal{O}$ under $C^\star$ would never trigger a warning from STACK, even if there might exist another C′ which would not allow eliminating that code. STACK's approach could be extended to support multiple interpretations to address this potential shortcoming.

5.1. A definition of unstable code

We now give a formal definition of unstable code. 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 $C^\star$ 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 $C^\star$ iff there exists a fragment e′ such that $\mathcal{P} \rightsquigarrow \mathcal{P}[e/e']$ is legal under C but* not *under $C^\star$.*

For example, for the sanity checks listed in Figure 3, a C compiler is entitled to replace them with *false*, as this is legal according to the C specification, whereas a hypothetical $C^\star$ compiler cannot do the same. Therefore, these checks are unstable code.

5.2. Algorithms 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 $\mathcal{O}$ under C but not under $C^\star$. In particular, STACK does this using a two-phase scheme:

1. run $\mathcal{O}$ without taking advantage of undefined behavior, which captures optimizations under $C^\star$; and
2. run $\mathcal{O}$ again, this time taking advantage of undefined behavior, which captures (more aggressive) optimizations under C.

If $\mathcal{O}$ optimizes extra code in the second phase, we assume the reason $\mathcal{O}$ did not do so in the first phase is because it would have changed the program's semantics under $C^\star$, 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 Section 5.2.1, 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 (Section 5.2.2) and one that simplifies unnecessary computation (Section 5.2.3).

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 $\mathbf{x}$, as summarized in Figure 4.

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

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

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* $\Delta(\mathbf{x})$*:*

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

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:

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

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 code 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.

Figure 4. Examples of C/C++ code fragments and their undefined behavior conditions. We describe their sufficient (though not necessary) conditions under which the code is undefined (Section J.2 in Ref.[23]). 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 `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; alias(p, q) predicates whether p and q point to the same object.

Code fragment	Sufficient condition	Undefined behavior
Core 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
Standard library:		
`abs`(x)	$x = -2^{n-1}$	Absolute value overflow
`memcpy(dst, src, len)`	`|dst − src| < len`	Overlapping memory copy
use q after `free`(p)	alias(p, q)	Use after free
use q after p' := `realloc`(p, ...)	alias(p, q) $\wedge$ $p' \neq \texttt{NULL}$	Use after realloc

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.

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* $\Delta(\mathbf{x})$*:*

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

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 nonconstant 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 4, 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:

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

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

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

Since there is no pointer p that satisfies this query, one can fold $p + 100 < p$ into *false*.

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. Similar to elimination, it warns if it finds e' that is equivalent to e only with the well-defined program assumption.

5.3. Implementation

We implemented STACK using the LLVM compiler framework[28] and the Boolector solver.[4] STACK consists of approximately 4000 lines of C++ code. To make the tool scale to large code bases, STACK implements an approximate version of the algorithms described in Section 5.2. Interested readers can refer to our SOSP paper for details.[44]

STACK focuses on identifying 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[41] or data races,[3] which STACK does not implement.

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

5.4. Main results

From July 2012 to March 2013, we periodically applied STACK to systems software written in C/C++, including 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 161 new bugs.

We also applied STACK to all 17,432 packages in the Debian Wheezy archive as of March 24, 2013. STACK checked 8575 of them that contained C/C++ code. Building and analyzing these packages took approximately 150 CPU-days on

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

```
1: procedure ELIMINATE($\mathcal{P}$)
2:     for all $e \in \mathcal{P}$ do
3:         if $R_e(\mathbf{x})$ is UNSAT then
4:             REMOVE($e$)                 ▷ trivially unreachable
5:         else
6:             if $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ is UNSAT then
7:                 REPORT($e$)
8:                 REMOVE($e$)         ▷ unstable code eliminated
```

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.

```
1: procedure SIMPLIFY($\mathcal{P}$, oracle)
2:     for all $e \in \mathcal{P}$ do
3:         for all $e' \in$ PROPOSE(oracle, $e$) do
4:             if $e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x})$ is UNSAT then
5:                 REPLACE($e$, $e'$)
6:                 break                 ▷ trivially simplified
7:             if $e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ is UNSAT then
8:                 REPORT($e$)
9:                 REPLACE($e$, $e'$)
10:                break         ▷ unstable code simplified
```

Intel Xeon E7-8870 2.4 GHz processors. For 3471 (40%) out of these 8575 packages, STACK issued at least one warning.

The results show that undefined behavior is widespread, and that STACK is useful for identifying undefined behavior. Please see our paper for more complete details.[44]

6. 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[25, 35, 42] and blog posts[27, 33, 34] 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.

6.1. 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[9] 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.

6.2. Optimization strategies

We believe that programmers should avoid undefined behavior. 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 Section 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.[8]

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`,[37] 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.[42]

6.3. Checkers

Many existing tools can detect undefined behavior as listed in Figure 4. For example, gcc provides the `-ftrapv` option to insert runtime checks for signed integer overflows (Section 3.18 in Ref.[36]); IOC[15] (now part of clang's sanitizers[12]) and KINT[43] cover a more complete set of integer errors; Saturn[16] finds null pointer dereferences; several dedicated C interpreters such as kcc[19] and Frama-C[10] perform checks for undefined behavior. See Chen et al.'s survey[11] 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.[16, 20] STACK, however, supports more expressive assumptions, such as pointer and integer operations.

As explored by existing checkers,[2, 21, 39] dead code is an effective indicator of likely bugs. STACK finds undefined behavior bugs by finding *subtly* unnecessary code under different interpretations of the language specification.

6.4. 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 in C (Section 6.5.7 in Ref.[23]), even though the result is consistently `0x80000000` on most modern processors. The committee for the C++ language standard has assigned well-defined semantics to this operation in the latest specification.[29]

7. SUMMARY

This paper demonstrates that undefined behavior bugs are much more prevalent than was previously believed, that they lead to a wide range of significant problems, that they are often misunderstood by system programmers, and that many popular compilers already perform unexpected optimizations, leading to misbehaving or vulnerable systems. We introduced a new approach for identifying undefined behavior, and developed a static checker, STACK, to help system programmers identify and fix bugs. We hope that compiler writers will also rethink optimization strategies against undefined behavior. Finally, we hope this paper encourages language designers to be careful with using undefined behavior in the language specification. Almost every language allows a developer to write programs that have undefined meaning according to the language specification. This research indicates that being liberal with what is undefined can lead to subtle bugs. All of Stack's source code is publicly available at http://css.csail.mit.edu/stack/.

Acknowledgments

We thank Xavier Leroy for helping improve this paper, and many others for their feedback on earlier papers.[42, 44] 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.

References

1. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM 53*, 2 (Feb. 2010), 66–75.
2. Blackshear, S., Lahiri, S. Almost-correct specifications: A modular semantic framework for assigning confidence to warnings. In *Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Seattle, WA, Jun. 2013), 209–218.
3. Boehm, H.-J. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Chicago, IL, Jun. 2005), 261–268.
4. Brummayer, R., Biere, A. 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* (York, UK, Mar. 2009), 174–177.
5. Bug 30475 – `assert(int+100 > int)` optimized away, 2007. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30475.
6. Bug 14287 – ext4: fixpoint divide exception at `ext4_fill_super`, 2009. https://bugzilla.kernel.org/show_bug.cgi?id=14287.
7. Bug 49820 – explicit check for integer negative after `abs` optimized away, 2011. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=49820.
8. Bug 53265 – warn when undefined behavior implies smaller iteration count, 2013. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=53265.
9. Cadar, C., Dunbar, D., Engler, D. 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).
10. Canet, G., Cuoq, P., Monate, B. A value analysis for C programs. In *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation* (Edmonton, Canada, Sept. 2009), 123–124.
11. Chen, H., Mao, Y., Wang, X., Zhou, D., Zeldovich, N., Kaashoek, M.F. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems* (Shanghai, China, Jul. 2011).
12. *Clang Compiler User's Manual: Controlling Code Generation*, 2014. http://clang.llvm.org/docs/UsersManual.html#controlling-code-generation.
13. Corbet, J. Fun with NULL pointers, part 1, July 2009. http://lwn.net/Articles/342330/.
14. Cuoq, P., Flatt, M., Regehr, J. Proposal for a friendly dialect of C, Aug. 2014. http://blog.regehr.org/archives/1180.
15. Dietz, W., Li, P., Regehr, J., Adve, V. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)* (Zurich, Switzerland, Jun. 2012), 760–770.
16. Dillig, I., Dillig, T., Aiken, A. Static error detection using semantic inconsistency inference. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (San Diego, CA, Jun. 2007), 435–445.
17. Dougherty, C.R., Seacord, R.C. C compilers may silently discard some wraparound checks. Vulnerability note VU#162289, US-CERT, 2008. http://www.kb.cert.org/vuls/id/162289, original version: http://www.isspcs.org/render.html?it=9100, also known as CVE-2008-1685.
18. Ellison, C., Roşu, G. *Defining the Undefinedness of C*. Technical report, University of Illinois, Apr. 2012. http://hdl.handle.net/2142/30780.
19. Ellison, C., Roşu, G. An executable formal semantics of C with applications. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL)* (Philadelphia, PA, Jan. 2012), 533–544.
20. Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B. 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)* (Chateau Lake Louise, Banff, Canada, Oct. 2001), 57–72.
21. Hoenicke, J., Leino, K.R.M., Podelski, A., Schäf, M., Wies, T. It's doomed; we can prove it. In *Proceedings of the 16th International Symposium on Formal Methods (FM)* (Eindhoven, the Netherlands, Nov. 2009), 338–353.
22. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2: Instruction Set Reference, A–Z*, Jan. 2013.
23. *ISO/IEC 9899:2011, Programming languages – C*, Dec. 2011.
24. Jack, B. Vector rewrite attack: Exploitable NULL pointer vulnerabilities on ARM and XScale architectures. White paper, Juniper Networks, May 2007.
25. Krebbers, R., Wiedijk, F. Subtleties of the ANSI/ISO C standard. Document N1639, ISO/IEC, Sept. 2012.
26. Lane, T. Anyone for adding `–fwrapv` to our standard CFLAGS? Dec. 2005. http://www.postgresql.org/message-id/1689.1134422394@sss.pgh.pa.us.
27. Lattner, C. What every C programmer should know about undefined behavior, May 2011. http://blog.llvm.org/2011/05/what-every-c-programmer-should-know.html.
28. Lattner, C., Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO)* (Palo Alto, CA, Mar. 2004), 75–86.
29. Miller, W.M. 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.
30. *Power ISA Version 2.06 Revision B, Book I: Power ISA User Instruction Set Architecture*, Jul. 2010.
31. Ranise, S., Tinelli, C., Barrett, C. QF_BV logic, Jun. 2013. http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2.
32. *Rationale for International Standard – Programming Languages – C*, Apr. 2003.
33. Regehr, J. A guide to undefined behavior in C and C++, Jul. 2010. http://blog.regehr.org/archives/213.
34. Regehr, J. Undefined behavior consequences contest winners, Jul. 2012. http://blog.regehr.org/archives/767.
35. Seacord, R.C. Dangerous optimizations and the loss of causality, Feb. 2010. https://www.securecoding.cert.org/confluence/download/attachments/40402999/Dangerous+Optimizations.pdf.
36. Stallman, R.M., the GCC Developer Community. *Using the GNU Compiler Collection for GCC 4.8.0*. GNU Press, Free Software Foundation, Boston, MA, 2013.
37. Teo, E. [PATCH] add `-fno-delete-null-pointer-checks` to gcc `CFLAGS`, Jul. 2009. https://lists.ubuntu.com/archives/kernel-team/2009-July/006609.html.
38. Tinnes, J. Bypassing Linux NULL pointer dereference exploit prevention (`mmap_min_addr`), Jun. 2009. http://blog.cr0.org/2009/06/bypassing-linux-null-pointer.html.
39. Tomb, A., Flanagan, C. Detecting inconsistencies via universal reachability analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, Jul. 2012), 287–297.
40. Torvalds, L. Re: [patch] CFS scheduler, -v8, May 2007. https://lkml.org/lkml/2007/5/7/213.
41. Tourrilhes, J. Invalid compilation without `-fno-strict-aliasing`, Feb. 2003. https://lkml.org/lkml/2003/2/25/270.
42. Wang, X., Chen, H., Cheung, A., Jia, Z., Zeldovich, N., Kaashoek, M.F. Undefined behavior: What happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems* (Seoul, South Korea, Jul. 2012).
43. Wang, X., Chen, H., Jia, Z., Zeldovich, N., Kaashoek, M.F. Improving integer security for systems with Kint. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI)* (Hollywood, CA, Oct. 2012), 163–177.
44. Wang, X., Zeldovich, N., Kaashoek, M.F., Solar-Lezama, A. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)* (Farmington, PA, Nov. 2013), 260–275.
45. Woods, J.F. Re: Why is this legal? Feb. 1992. http://groups.google.com/group/comp.std.c/msg/dfe1ef367547684b.

Xi Wang ({xi}@cs.washington.edu), University of Washington, Seattle, WA.

Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama ({nickolai, kaashoek, asolar}@csail.mit.edu), Massachusetts Institute of Technology, Cambridge, MA.