A Differential Approach to Undefined Behavior Detection

by

Xi Wang

B.E., Computer Science (2005); M.E., Computer Science (2008)
Tsinghua University

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

Signature redacted

Author....... ....
Department of Electrical Engineering and Computer Science
August 29, 2014

Signature redacted

Certified by............ ....
M. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

Signature redacted

Certified by........ ....
Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Signature redacted

Accepted by ....
Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

A Differential Approach to Undefined Behavior Detection

by

Xi Wang

Submitted to the Department of Electrical Engineering and Computer Science
on August 29, 2014, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis 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 thesis 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.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor

Thesis Supervisor: Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

I owe an immeasurable debt to my advisors, Frans Kaashoek and Nickolai Zeldovich, for their constant guidance, enthusiastic support, and extraordinary patience. Their huge impact on my life is beyond words. I would like to express my sincere gratitude to Robert Morris for his endless source of wisdom, and to Armando Solar-Lezama for his invaluable insight. This thesis would not have been possible without their gracious advice and encouragement.

This research was greatly improved by the feedback of Adam Chlipala, Austin Clements, Victor Costan, John Guttag, Eddie Kohler, Xuezheng Liu, Li Lu, Shan Lu, Neha Narula, John Regehr, Jesse Ruderman, Linchun Sun, Frank Wang, Tom Woodfin, Junfeng Yang, Lidong Zhou, and the anonymous reviewers of APSys 2012 and SOSP 2013.

Nickolai and Haogang Chen made substantial contributions to the implementation of STACK. STACK reused various components written by Zhihao Jia from KINT [58], and by Yandong Mao and Dong Zhou from LXFI [40]. John Regehr's blog and Haohui Mai's experience with SMT solvers were sources of inspiration for this work.

I have had the incredible fortune to work with amazing friends and colleagues in the PDOS group at MIT and the systems research group at Microsoft Research Asia. I have enjoyed every minute with them.

Last but not least, I thank my parents and my soulmate, X. K., for their unconditional love and fantastic cooking skills.

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.

Portions of this thesis were published in:

[57] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*, Seoul, South Korea, July 2012.

[59] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 260–275, Farmington, PA, November 2013.

Contents

List of Figures

Chapter 1

Introduction

Undefined behavior in systems programming languages is a dark side of systems programming. It introduces unpredictable systems behavior, and has a significant impact on reliability and security. This thesis proposes a new approach that identifies undefined behavior by finding code fragments that have divergent behavior under different interpretations of the language specification. It is scalable, precise, and practical: the Stack checker that implements this approach has uncovered 161 new bugs in real-world software. This chapter introduces the problem, and provides an overview of our approach and tool.

1.1 Undefined behavior

The specifications of many programming languages designate certain code fragments as having *undefined behavior* [18: §2.3]. For instance, in C "use of a nonportable or erroneous program construct or of erroneous data" leads to undefined behavior [30: §3.4.3]. A comprehensive list of undefined behavior in C is available in the language specification [30: §J.2].

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 [27: §3.2], while it silently produces an undefined result on PowerPC [25: §3.3.8]. 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.

```
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 1-1: A pointer overflow check found in several code bases. The code becomes vulnerable as gcc optimizes away the second `if` statement [16].

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., run-time 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.

1.2 Risks of 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" [61]. But what happens in practice?

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, a simple SQL 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 (see §5.2.1 for details).

In addition, compiler optimizations can amplify the effects of undefined behavior. For example, consider the pointer overflow check `buf + len < buf` shown in Figure 1-1, 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 1-1. We have found similar checks in a number of systems, including the Chromium browser [8], the Linux kernel [57], and the Python interpreter [45].

While this check appears to work with a flat address space, it fails on a segmented architecture [29: §6.3.2.3]. Therefore, the C standard states that an overflowed pointer is undefined [30: §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 [16].

As we will demonstrate in §2, many optimizing compilers make similar assumptions that programmers never invoke undefined behavior. Consequently, these compilers turn each operation into an assumption about the arguments to that operation. The compilers then proceed to optimize the rest of the program under these assumptions.

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" [54] and merely the compiler's "creative reinterpretation of basic C semantics" [33]. On the other hand, compiler writers argue that the optimizations are legal under the specification; it is the "broken code" [21] 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 §2.2, many compilers have become more aggressive over the past 20 years with such optimizations.

1.3 Status and 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 §2.1, 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 §2.2, 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 [13, 41], which has less undefined behavior, though in general it remains unclear how to outlaw undefined behavior from the specification without incurring significant performance overhead [13, 57].

Certain static-analysis and model checkers identify classes of bugs due to undefined behavior. For example, tools that find buffer overflow bugs [7] can be viewed as finding undefined behavior bugs, because referencing a location outside of a buffer's range is undefined behavior. Our goal is to go after bugs due to undefined behavior that stem from a programmer's misunderstanding of the language specification (see §6 for details).

1.4 Approach: finding divergent behavior

Ideally, compilers would generate warnings for developers when an application invokes undefined behavior, and this thesis 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 [2].

The local likely invariant that we exploit in this thesis 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 thesis, 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′.

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 thesis, 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.

1.5 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 [46]. 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 $\mathbb{O}$ to implicitly flag unnecessary code. STACK's $\mathbb{O}$ eliminates dead code and performs expression simplifications under the semantics of C and $C^\star$, respectively. For code fragment e, if $\mathbb{O}$ is *not* able to rewrite e under neither semantics, STACK considers e as "live code"; if $\mathbb{O}$ is able to rewrite e under both semantics, e is "dead code"; if $\mathbb{O}$ is able to rewrite e under C but not $C^\star$, STACK reports it as "unstable code." We describe this approach more precisely in §3.

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 $\mathbb{O}$ 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.

1.6 Contributions

This thesis makes several contributions, as follows:

1. The first detailed study of the impact and prevalence of undefined behavior bugs in

real-world software, and of how compilers amplify the problems. This study finds that undefined behavior is prevalent, has many risks, and is increasingly exploited by compiler optimizations.

2. A scalable approach to detecting undefined behavior in large programs through differential interpretation.
3. A formalization of this approach that can be applied in practice.
4. A practical static analysis tool, called Stack, based on this formalization.
5. A large-scale evaluation of Stack, which demonstrates that Stack can find 161 real bugs in a wide range of widely used software. We reported these bugs to developers, and almost all of them were fixed, suggesting that Stack's reports are precise.

Overall, this thesis demonstrates that undefined behavior bugs are much more prevalent than was previously believed and that they lead to a wide range of significant problems.

1.7 Roadmap

The rest of the thesis is organized as follows. §2 provides a detailed case study of unstable code in real systems and compilers. §3 presents a formalization of unstable code. §4 describes the design and implementation of the Stack checker for identifying unstable code. §5 reports our experience of applying Stack to identify unstable code and evaluates Stack's techniques. §6 covers related work. §7 concludes.

Chapter 2

Case studies

This chapter provides case studies of undefined behavior and how it can lead to unstable code. It builds on earlier surveys [32, 49, 57] and blog posts [35, 47, 48] that describe unstable code examples, and extends them by investigating the evolution of optimizations in compilers. From the evolution we conclude that unstable code will grow as future compilers implement more aggressive optimization algorithms.

2.1 Examples of unstable code

It is well known that compiler optimizations may produce undesirable behavior for imperfect code [24]. Recent advances in optimization techniques further increase the likelihood of such undesired consequences, as they exploit undefined behavior aggressively, exposing unstable code as a side effect. This section reviews representative cases that have sparked interest among programmers and compiler writers.

2.1.1 Pointer overflow and a disputed vulnerability note

As described in §1.2, the C language standard states that an overflowed pointer is undefined [30: §6.5.6/p8], which voids any pointer "overflow" check, such as the check `buf + len < buf` shown in Figure 1-1. This allows the compiler to perform aggressive optimizations, including removing the check.

The earliest report of such an optimization that we are aware of is a gcc bug filed in 2006, in which a programmer reported that gcc removed a pointer overflow check intended for

```
unsigned int tun_chr_poll(struct file *file, poll_table * wait)
{
    struct tun_file *tfile = file->private_data;
    struct tun_struct *tun = __tun_get(tfile);
    struct sock *sk = tun->sk;
    if (!tun)
        return POLLERR;
    ...
}
```

Figure 2-1: A null pointer dereference vulnerability (CVE-2009-1897). The dereference of pointer tun comes before the null pointer check. The code becomes exploitable as gcc optimizes away the null pointer check [11].

validating network packets, even "without optimizer" (i.e., using -O0) [20]. This bug was marked as a "duplicate" with no further action.

The issue received much attention in 2008 when the US-CERT published a vulnerability note regarding a crash bug in plan9port (Plan 9 from User Space), suggesting programmers "avoid newer versions of gcc" in the original security alert [16]. The crash was caused by gcc removing a pointer overflow check in a string formatting function [12]. The gcc developers disputed the vulnerability note and argued that this optimization is allowed by the specification and performed by many other compilers as well. The vulnerability note was revised later, with "gcc" changed to "some C compilers" [16].

2.1.2 Null pointer dereference and a kernel exploit

In addition to introducing new vulnerabilities, optimizations that remove unstable code can amplify existing weaknesses in the system. Figure 2-1 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 [31, 53]), 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, this simple bug becomes an exploitable vulnerability. 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 [30: §6.5.3]. Since tun is

```
int do_fallocate(..., loff_t offset, loff_t len)
{
    struct inode *inode = ...;
    if (offset < 0 || len <= 0)
        return -EINVAL;
    /* Check for wrap through zero too */
    if ((offset + len > inode->i_sb->s_maxbytes)
        || (offset + len < 0))
        return -EFBIG;
    ...
}
```

Figure 2-2: A signed integer overflow check `offset + len < 0`. The intention was to prevent the case when `offset + len` overflows and becomes negative.

non-null, gcc further determines that the null pointer check is unnecessary and eliminates the check, making a privilege escalation exploit possible that otherwise would not be [11].

2.1.3 Signed integer overflow from day one

Signed integer overflow has been present in C even before there was a standard for the language—the Version 6 Unix used the check `mpid + 1 < 0` to detect whether it runs out of process identifiers, where `mpid` is a non-negative counter [38: §7.13]. Such overflow checks are unstable code and unlikely to survive with today's optimizing compilers. For example, both gcc and clang conclude that the "overflow check" $x + 100 < x$ with a signed integer x is always false. Some programmers were shocked that gcc turned the check into a no-op, leading to a harsh debate between the C programmers and the gcc developers [21].

A common misbelief is that signed integer operations always silently wrap around on overflow using two's complement, just like unsigned operations. This is false at the instruction set level, including older mainframes that use one's complement, embedded processors that use saturation arithmetic, and even architectures that use two's complement. For example, while most x86 signed integer instructions do silently wrap around, there are exceptions, such as signed division that traps for `INT_MIN`$/-1$ [28: §3.2]. In C, signed integer overflow is undefined behavior [30: §6.5].

Figure 2-2 shows another example from the `fallocate` system call implementation in the Linux kernel. Both `offset` and `len` are provided by a user-space application; they cannot be trusted and must be validated by the kernel. Note that they are of the signed integer type `loff_t`.

```
struct timeval tv;
unsigned long junk;      /* XXX left uninitialized on purpose */
gettimeofday(&tv, NULL);
srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
```

Figure 2-3: An uninitialized variable misuse for pseudorandom number generation. It was in the libc of FreeBSD and OS X; clang optimizes away the entire seed computation (CVE-2013-5180).

The code first rejects negative values of `offset` and `len`, and checks whether `offset + len` exceeds some limit. The comment says "[c]heck for wrap through zero too," indicating that the programmer realized that the addition may overflow and bypass the limit check. The programmer then added the overflow check `offset + len < 0` to prevent the bypass.

However, gcc is able to infer that both `offset` and `len` are non-negative at the point of the overflow check. Along with the knowledge that signed addition overflow is undefined, gcc concludes that the sum of two non-negative integers must be non-negative. This means that the check `offset + len < 0` is always false and gcc removes it. Consequently, the generated code is vulnerable: an adversary can pass in two large positive integers from user space, the sum of which overflows, and bypass all the sanity checks.

2.1.4 Uninitialized read and less randomness

A local variable in C is *not* initialized to zero by default. A misconception is that such an uninitialized variable lives on the stack, holding a "random" value. This is not true. A compiler may assign the variable to a register (e.g., if its address is never taken), where its value is from the last instruction that modified the register, rather than from a stack location. Moreover, on Itanium if the register happens to hold a special not-a-thing value, reading the register traps except for a few instructions [26: §3.4.3].

Reading an uninitialized variable is undefined behavior in C [30: §6.3.2.1]. A compiler can assign any value to the variable and also to expressions derived from the variable.

Figure 2-3 shows such a problem in the `srandomdev` function of FreeBSD's libc, which also appears in DragonFly BSD and Mac OS X. The corresponding commit message says that the programmer's intention of introducing `junk` was to "use stack junk value," which is left uninitialized intentionally, as a source of entropy for pseudorandom number generation. Along with current time from `gettimeofday` and the process identification from `getpid`, the code computes a seed value for `srandom`.

Unfortunately, the use of `junk` does not introduce more randomness from the stack: both

	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	–	–	–	–	–

Figure 2-4: 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, "On" 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.

gcc and clang assign `junk` to a register; clang further eliminates computation derived from `junk` completely, and generates code that does *not* use either `gettimeofday` or `getpid`.

2.2 An evolution of optimizations

To understand the evolution of compilers with respect to optimizing unstable code, we conduct a study using six representative examples in the form of sanity checks, as shown in the top row of Figure 2-4. 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 1-1 in §2.1.1.
- The null pointer check $!p$ with an earlier dereference is from Figure 2-1 in §2.1.2.
- The check $x + 100 < x$ with a signed integer x is from §2.1.3.
- Another 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 [37].

- The check abs$(x) < 0$, intended to catch the most negative value (i.e., -2^{n-1}), tests whether optimizations understand library functions [22].

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, we find the lowest optimization level `-O`*n* at which it happens. The result is shown in Figure 2-4.

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 4.9.1, the latest version, released in 2014.

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

We make the following observations of existing compilers from Figure 2-4. First, eliminating unstable code is common among compilers, not just in recent gcc versions as some programmers have claimed [33]. Even gcc 2.95.3 eliminates $x + 100 < x$. Some compilers discard unstable code that gcc does not (e.g., clang on $1 \ll x$).

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 [44].

Third, discarding unstable code occurs with standard optimization options, mostly at `-O2`, the default optimization level for release builds (e.g., `autoconf` [39: §5.10.3]); some compilers even discard unstable code at the lowest level of optimization `-O0`. Hence, lowering the optimization level as Postgres did [34] is an unreliable way of working around unstable code.

Fourth, optimizations even exploit undefined behavior in library functions (e.g., `abs` [22] and `realloc` [48]) 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.

Chapter 3

Formalizing 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 [47: §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 §5.2, 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 A definition of unstable code

To formalize a programmer's misunderstanding of the C specification that leads to unstable code, let $\mathrm{C}^\star$ denote a C dialect that assigns well-defined semantics to code fragments that have undefined behavior in C. For example, $\mathrm{C}^\star$ is defined for a flat address space, a null pointer that maps to address zero, and wrap-around semantics for pointer and integer arithmetic [46]. 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 $\mathrm{C}^\star$ 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* 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 2-4, 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 $\mathcal{O}$ under C but not under C*. 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*; 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*, 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

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
$x << y$, $x >> y$	$y < 0 \lor y \geq n$	oversized shift
$a[x]$	$x < 0 \lor x \geq \texttt{ARRAY\_SIZE}(a)$	buffer overflow
standard library:		
abs(x)	$x = -2^{n-1}$	absolute value overflow
memcpy(dst, src, len)	$\lvert\texttt{dst} - \texttt{src}\rvert < \texttt{len}$	overlapping memory copy
use q after free(p)	alias(p, q)	use after free
use q after $p' :=$ realloc($p, \ldots$)	alias(p, q) $\land\ p' \neq \texttt{NULL}$	use after realloc

Figure 3-1: 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 [30: §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 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.

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 $\mathbf{x}$, as summarized in Figure 3-1.

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:

$$R_e(\mathbf{x}) \to \neg U_e(\mathbf{x}). \tag{3.1}$$

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

$$\Delta(\mathbf{x}) = \bigwedge_{e \in \mathcal{P}} R_e(\mathbf{x}) \to \neg U_e(\mathbf{x}). \tag{3.2}$$

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

$$\nexists \mathbf{x} \colon R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}). \tag{3.3}$$

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 2-1 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

```
procedure ELIMINATE($\mathcal{P}$)
    for all $e \in \mathcal{P}$ do
        if $R_e(\mathbf{x})$ is UNSAT then
            REMOVE($e$)                                  ▷ trivially unreachable
        else
            if $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ is UNSAT then
                REPORT($e$)
                REMOVE($e$)                              ▷ unstable code eliminated
```

Figure 3-2: The elimination algorithm. It reports unstable code that becomes unreachable with the well-defined program assumption.

which is *true*. As a result, the elimination query $R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ for the `return` statement is:

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

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 3-2). 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.

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

$$\exists e' \nexists \mathbf{x}\colon\, e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}). \tag{3.4}$$

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

```
procedure SIMPLIFY($\mathcal{P}$, oracle)
    for all $e \in \mathcal{P}$ do
        for all $e' \in$ PROPOSE(oracle, $e$) do
            if $e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x})$ is UNSAT then
                REPLACE($e$, $e'$)
                break                                              ▹ trivially simplified
            if $e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R_e(\mathbf{x}) \wedge \Delta(\mathbf{x})$ is UNSAT then
                REPORT($e$)
                REPLACE($e$, $e'$)
                break                                              ▹ unstable code simplified
```

Figure 3-3: 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.

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 3-1, 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*. §5.2.2 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 3-3). 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 [55] or data races [3], 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.

Chapter 4

The STACK checker

This chapter describes the design and implementation 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 4-1. 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 3-1 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 2-1 STACK links the null pointer check `!tun` to the earlier pointer dereference `tun->sk`.

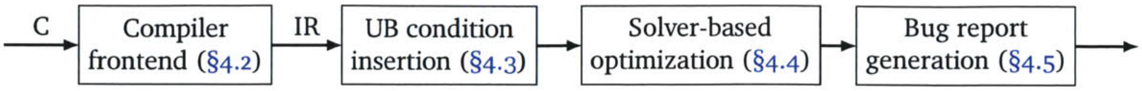
C
Compiler frontend (§4.2)
IR
UB condition insertion (§4.3)
Solver-based optimization (§4.4)
Bug report generation (§4.5)

Figure 4-1: Stack's workflow. It invokes clang to convert a C/C++ program into LLVM IR, and then detects unstable code based on the IR.

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 3-1. 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: 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 [4] to decide satisfiability for elimination and simplification queries, as shown in (3.3) and (3.4). Both queries need to compute the terms $R_e(\mathbf{x}) \land \Delta(\mathbf{x})$. 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 dom(e) denote e's dominators [43: §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 dom(e).

```
procedure MinUBCond($Q_e\,[= H \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x})]$)
    $ubset \leftarrow \varnothing$
    for all $d \in \mathrm{dom}(e)$ do
        $Q'_e \leftarrow H \wedge \bigwedge_{d' \in \mathrm{dom}(e) \setminus \{d\}} \neg U_{d'}(\mathbf{x})$
        if $Q'_e$ is SAT then
            $ubset \leftarrow ubset \cup \{U_d\}$
    return $ubset$
```

Figure 4-2: Algorithm for computing the minimal set of UB conditions. These UB conditions lead to unstable code given query Q_e for fragment 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:

$$R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{4.1}$$

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

$$e(\mathbf{x}) \neq e'(\mathbf{x}) \wedge R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{4.2}$$

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 [56]. 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.

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 [9], 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 (4.1) and (4.2), we know that the query must be in the following form:

$$Q_e = H \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{4.3}$$

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 4-2.

4.6 Limitations

The list of undefined behavior Stack implements (see Figure 3-1) is incomplete. For example, it misses violations of strict aliasing [30: §6.5] and uses of uninitialized variables [30: §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" [30: §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, Stack will also miss unstable code if the solver times out. See §5.6 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 §5.2.4). Note that even though such redundant code fragments are false warnings, discarding them is allowed by the specification.

4.7 Implementation

We implemented STACK using the LLVM compiler framework [36] and the Boolector solver [4]. STACK consists of approximately 4,000 lines of C++ code.

Chapter 5

Evaluation

This chapter answers the following questions:

- Is STACK useful for finding new bugs? (§5.1)
- What kinds of unstable code does STACK find? (§5.2)
- How precise are STACK's bug reports? (§5.3)
- How long does STACK take to analyze a large system? (§5.4)
- How prevalent is unstable code in real systems, and what undefined behavior causes it? (§5.5)
- What unstable code does STACK miss? (§5.6)

5.1 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 5-1, 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 161 new bugs. The results show that unstable code is widespread, and that STACK is useful for identifying unstable code.

Figure 5-1 also breaks down the bugs by type of undefined behavior. The results show that several kinds of undefined behavior contribute to the unstable code bugs.

	# 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 [62]	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	3		1	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	161	29	44	24	7	23	14	1	7	9	3

(★) Bionic, Dune [1], file, GMP, Mosh [60], MySQL, OpenSSH, OpenSSL, PHP, Wireshark.

Figure 5-1: New bugs identified by STACK. We also break down the number of bugs by undefined behavior from Figure 3-1: "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).

5.2 Analysis of bug reports

This section reports our experience of finding and fixing unstable code with the aid of STACK. We manually 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 §2.2 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 section 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.

5.2.1 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 2-1, 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 5-2. 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*.

```
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 5-2: An invalid signed division overflow check in Postgres. Note that 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 5-3: An incorrect null pointer check in Linux's sysctl. A correct null check should test the result of `strchr`, rather than that plus one, which is always non-null.

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" [42]. 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 designed 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 §5.2.3.

```
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 5-4: Unstable bounds checks in the form data + x < data from FFmpeg/Libav. For these checks, gcc optimizes them into $x < 0$.

By identifying unstable code, Stack is also useful for uncovering programming errors that do not directly invoke undefined behavior. Figure 5-3 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.

5.2.2 Urgent optimization bugs

Urgent optimization bugs are unstable code that existing compilers already optimize to cause problems. §2.1 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 5-4. The parsing code starts 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.

```
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 5-5: 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.

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

Figure 5-6: A time bomb in Postgres. The intention is to check whether arg1 is the most negative value -2^{n-1}, similar to Figure 5-5.

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 1-1: 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.

Figure 5-5 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

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

Figure 5-7: Redundant code from the Linux kernel. The caller of this code snippet ensures that c must be non-null and the null pointer check against c is always *true*.

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.

5.2.3 Time bombs

A time bomb is unstable code that is harmless at present, since no compiler listed in §2.2 can currently optimize it. But this situation may change over time. §2.2 already showed how past compiler changes trigger time bombs to become urgent optimization bugs. §5.2.1 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 5-6 shows a time bomb example from Postgres. As mentioned in §5.2.1, 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 5-5. 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 [51] already discard the check. Relying on compilers to not optimize time bombs for system security is risky, and we recommend fixing problems flagged by STACK to avoid this risk.

5.2.4 Redundant code

Figure 5-7 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

	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 5-8: Stack's performance of analyzing Kerberos, Postgres, and the Linux kernel. This includes 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.

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 §5.3).

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.

5.3 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 §5.2.1. 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.

Stack found 26 time bombs (see §5.2.3 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 5-9: Number of reports generated by each of Stack's algorithms. This is from §3.2 for all Debian Wheezy packages. We also include the number of packages for which at least one such report was generated.

5.4 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 5-8 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.

5.5 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 5-9 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.

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 5-10: Number of reports that involve each of Stack's UB conditions. This is from Figure 3-1 for all Debian Wheezy packages. We also include the number of packages for which at least one such report was generated.

Figure 5-10 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.

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.

5.6 Completeness

Stack is able to identify all the unstable code examples described in §2.2. 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 [48] and Wang et al.'s undefined behavior survey [57] 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.

Chapter 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 [32, 49, 57] and blog posts [35, 47, 48] 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 [5] 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 thesis.

6.2 Optimization strategies

We believe that programmers should avoid undefined behavior, and we provide suggestions for fixing unstable code in §5.2. 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 [23].

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` [52], 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 [57].

6.3 Checkers

Many existing tools can detect undefined behavior as listed in Figure 3-1. For example, gcc provides the `-ftrapv` option to insert run-time checks for signed integer overflows [50: §3.18];

IOC [14] (now part of clang's sanitizers [10]) and KINT [58] cover a more complete set of integer errors; Saturn [15] finds null pointer dereferences; several dedicated C interpreters such as kcc [17] and Frama-C [6] perform checks for undefined behavior. See Chen et al.'s survey [7] 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 [15, 19]. STACK, however, supports more expressive assumptions, such as pointer and integer operations.

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 [30: §6.5.7], 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 [41].

Chapter 7

Conclusion

This thesis 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 approach 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 thesis 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 STACK source code is publicly available at `http://css.csail.mit.edu/stack/`.

Appendix 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., $Q \to Q'$): 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 (4.1) is weaker than the original query (3.3); the simplification queries (4.2) and (3.4) are similar. Formally, given code fragment e, it suffices to show the following:

$$R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}) \to R'_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{A.1}$$

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:

$$\Delta(\mathbf{x}) \to \bigwedge_{d \in \mathrm{dom}(e)} \big(R_d(\mathbf{x}) \to \neg U_d(\mathbf{x})\big). \tag{A.2}$$

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

$$\forall d \in \mathrm{dom}(e)\colon R_e(\mathbf{x}) \to R_d(\mathbf{x}). \tag{A.3}$$

Combining (A.2) and (A.3) gives:

$$\Delta(\mathbf{x}) \to \big(R_e(\mathbf{x}) \to \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x})\big). \tag{A.4}$$

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

$$R_e(\mathbf{x}) \wedge \Delta(\mathbf{x}) \to R_e(\mathbf{x}) \wedge \bigwedge_{d \in \mathrm{dom}(e)} \neg U_d(\mathbf{x}). \tag{A.5}$$

By definition $R_e(\mathbf{x}) \to R'_e(\mathbf{x})$, so (A.5) implies (A.1). □

Bibliography

[1] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos 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, October 2012.

[2] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2): 66–75, February 2010.

[3] Hans-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.

[4] Robert Brummayer and Armin 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, March 2009.

[5] Cristian Cadar, Daniel Dunbar, and Dawson 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, December 2008.

[6] Géraud Canet, Pascal Cuoq, and Benjamin 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, September 2009.

[7] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans 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.

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

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

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

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

[12] Russ Cox. Re: plan9port build failure on Linux (debian), March 2008. http://9fans.net/archive/2008/03/89.

[13] Pascal Cuoq, Matthew Flatt, and John Regehr. Proposal for a friendly dialect of C, August 2014. http://blog.regehr.org/archives/1180.

[14] Will Dietz, Peng Li, John Regehr, and Vikram 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.

[15] Isil Dillig, Thomas Dillig, and Alex 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.

[16] Chad R. Dougherty and Robert 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, original version http://www.isspcs.org/render.html?it=9100, also known as CVE-2008-1685.

[17] Chucky Ellison and Grigore Rou. 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, January 2012.

[18] Chucky Ellison and Grigore Rou. Defining the undefinedness of C. Technical report, University of Illinois, April 2012. http://hdl.handle.net/2142/30780.

[19] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin 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, October 2001.

[20] GCC. Bug 27180 - pointer arithmetic overflow handling broken, 2006. http://gcc.gnu.org/bugzilla/show_bug.cgi?id=27180.

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

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

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

[24] IBM. Optimizing C code at optimization level 2. White paper, September 2009.

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

[26] Intel. *Intel Itanium Architecture Software Developer's Manual, Volume 1: Application Architecture*, May 2010.

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

[28] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2014.

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

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

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

[32] Robbert Krebbers and Freek Wiedijk. Subtleties of the ANSI/ISO C standard. Document N1639, ISO/IEC JTC1/SC22/WG14, September 2012.

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

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

[35] Chris 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.

[36] Chris Lattner and Vikram 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, March 2004.

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

[38] John Lions. *A Commentary on the Sixth Edition UNIX Operating System*. 1977.

[39] David MacKenzie, Ben Elliston, and Akim Demaille. *Autoconf: Creating Automatic Configuration Scripts for version 2.69*. Free Software Foundation, April 2012.

[40] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M. Frans Kaashoek. Software fault isolation with API integrity and multi-principal modules. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 115–128, Cascais, Portugal, October 2011.

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

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

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

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

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

[46] Silvio Ranise, Cesare Tinelli, and Clark Barrett. QF_BV logic, 2013. http://smtlib.cs.uiowa.edu/logics/QF_BV.smt2.

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

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

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

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

[51] Mark Stephenson, Jonathan Babb, and Saman 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.

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

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

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

[55] Jean Tourrilhes. Invalid compilation without -fno-strict-aliasing, February 2003. https://lkml.org/lkml/2003/2/25/270.

[56] Peng Tu and David 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.

[57] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Undefined behavior: What happened to my code? In *Proceedings of the 3rd Asia-Pacific Workshop on Systems*, Seoul, South Korea, July 2012.

[58] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans 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, October 2012.

[59] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. Towards optimization-safe systems: Analyzing the impact of undefined behavior. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 260–275, Farmington, PA, November 2013.

[60] Keith Winstein and Hari 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.

[61] John F. Woods. Re: Why is this legal?, February 1992. http://groups.google.com/group/comp.std.c/msg/dfe1ef367547684b.

[62] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David 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, November 2006.