Modular Bug-finding for Integer Overflows in the Large:
Sound, Efficient, Bit-precise Static Analysis

Technical Report
MSR-TR-2009-57

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Modular Bug-finding for Integer Overflows in the Large: Sound, Efficient, Bit-precise Static Analysis

Yannick Moy[*], Nikolaj Bjørner, and David Sielaff
Microsoft Research, Redmond, Washington, USA
yannick.moy@gmail.com, {nbjorner, dsielaff}@microsoft.com

Abstract

We describe a methodology and a tool for performing scalable bit-precise static analysis. The tool combines the scalable static analysis engine PREfix [14] and the bit-precise efficient SMT solver Z3 [20]. Since 1999, PREfix has been used at Microsoft to analyze C/C++ production code. It relies on an efficient custom constraint solver, but addresses bit-level semantics only partially. On the other hand, the Satisfiability Modulo Theories solver Z3, developed at Microsoft Research, supports precise machine-level semantics for integer arithmetic operations.

The integration of PREfix with Z3 allows uncovering software bugs that could not previously be identified. Of particular importance are integer overflows. These typically arise when the programmer wrongly assumes mathematical integer semantics, and they are notorious causes of buffer overflow vulnerabilities in C/C++ programs.

We performed an experimental evaluation of our integration by running the modified version of PREfix on a large legacy code base for the next version of a Microsoft product. The experiments resulted in a number of bugs filed and fixed related to integer overflows. We also describe how we developed useful filters for avoiding false positives based on the comprehensive evaluation.

1 Introduction

Integer overflows have received heightened attention recently due to the increasing number of reports of security vulnerabilities where they can be exploited. The main problem with integer overflows is that they can manifest in subtle ways in any program that manipulates integers, which is to say every program at all. Figure 1 presents commonly found implementations in C of well-known algorithms for searching in a sorted array and converting an integer to ASCII. Both functions are in fact vulnerable to integer

```
int binary_search(
  int* arr, int low, int high, int key)
{
    while (low <= high)
    {
        // Find middle value
        int mid = (low + high) / 2;
        int val = arr[mid];
        // Refine range
        ...
}
```

```
void itoa(int n, char* buf)
{
    // Handle negative
    if (n < 0)
    {
        *buf++ = '-';
        n = -n;
    }
    // Output digits
    ...
}
```

Figure 1: Some classical programs suffering from integer overflows

overflows: in the case of function `binary_search` applied to `low` and `high` both equal to $\frac{\texttt{INT_MAX+1}}{2}$ (0x40000000), the computation `low + high` will evaluate to `INT_MIN` instead of $\frac{\texttt{INT_MAX+1}}{2}$; in the

[*]Work performed while at Microsoft Research

```
uint rest(
  uint sz, uint done)
{
  return sz - done;
}
```

```
void alloc(
  uint pos)
{
  int sz = f();
  if (pos < sz) ...
}
```

Figure 2: Integer overflows: arithmetic and cast

case of function `itoa` applied to `n` equal to `INT_MIN` (0x80000000), the negation `-n` will evaluate to `INT_MIN` instead of `INT_MAX+1`. Despite the publicity given to the case of binary search [8], many such programs remain vulnerable to integer overflows. The case of `itoa` is compelling: the first edition of *The C Programming Language* in 1978 [34] contained the integer oveflow problem just mentioned; the problem was noted in the second edition in 1988 (and its solution left in exercise), but many currently available implementations, such as the one from project *itoa* on *sourceforge.net*, still suffer from the same problem.

1.1 Integer Overflows in Practice

Some dynamically typed languages like LISP, Python and JavaScript prevent integer overflows by using *bignums* as the default integer representation. Efficient implementations of bignums use machine integers for small numbers and switch to general, and more expensive, representations when the operations on machine integers overflow. Most languages, however, do not support this default semantics and common programs will be exposed to integer overflows when working with machine integers. Integer overflows occur when the result of an arithmetic operation on machine integers is different from the mathematical result of the operation. In C/C++, this problem is exacerbated by the incredible number of machine integer types (6 on 32-bits machines, 8 on 64-bits machines), which can be freely mixed in operations and converted one to another, with subtle rules that can trick even experts (*e.g.*, see conclusion in [40]).

As a result, machine integer semantics are often overlooked by programmers who wrongly assume mathematical integers semantics for values that are outside of the range where the two semantics coincide. Bugs related to integer overflows are of two kinds, according to the operation that leads to the overflow: arithmetic and cast. Figure 2 shows an example of code for each kind. Function `rest` may be called with `sz < done`, in which case the result of the subtraction is a large positive integer. Function `alloc` can end up executing the `then` branch if `sz` is negative, due to the implicit cast of `sz` to unsigned integer performed in the test `pos < sz`.

One difficulty in finding integer overflow bugs stems from the fact that integer overflows are perfectly legal in C/C++. While the C and C++ standards [30, 31] distinguish cases in which integer overflows follow a modulo semantics (*e.g.*, arithmetic on unsigned integers) from cases where integer overflows trigger undefined behavior (*e.g.*, arithmetic on signed integers), most architectures give a modulo semantics to all integer overflows. Many programs rely on such behavior. Therefore it is not possible to discard integer overflows when analyzing these programs. From our experience at looking for integer overflow bugs in Microsoft code base, there are three main cases where integer overflows are intended. Figure 3 shows an example of each.

- Overflow is intentional. Typically, this is almost always the case when the programmer has inserted an explicit cast in the code. *E.g.*, this is the case in function `htons`, where the higher bits of `x` should be ignored.

```
uint16 htons(
  uint16 a)
{
  uint x = a << 8;
  uint y = a >> 8;
  return (uint16)(x | y);
}
```

```
uint i = 0;
while (i < max) {
  if (sel(vect[i])) {
    vect.remove(i);
    --i;
  }
  ++i;
}
```

```
uint safe_uadd(
  uint a, uint b)
{
  uint r = a + b;
  if (r < a) error();
  return r;
}
```

Figure 3: Expected integer overflows: intentional, reversed and checked

- Overflow is reversed *a posteriori*. This is usually associated to increments or decrements performed in a loop. *E.g.*, when an element is removed from vector `vect`, index `i` is decremented to compensate for the increment at the end of the loop. If the element at index `0` is removed, then unsigned integer `i` overflows to `UINT_MAX`, but the subsequent increment reverses the overflow.

- Overflow is checked *a posteriori*. This is either done in dedicated functions that perform safe operations, like `safe_uadd`, or inlined where required. In the case of `safe_uadd`, the test `r < a` indeed filters all cases where an overflow occurs in the operation `a + b`.

So, we are interested in finding bugs related to integer overflows in large code bases written in C/C++. Among these, we are mostly interested in finding those bugs that can lead to a buffer overflow that an attacker could exploit to craft an elevation of privilege (EoP) attack, or a denial of service (DoS) attack. Ultimately, like in most bug-finders, our tool depends on a human reviewer to decide the inocuity or severity of the integer overflows reported. This is all the more the case with integer overflows due to the subtlety of the associated bugs.

Our constraints are thus the following: (1) there are no manual annotations for integer overflows in the code, (2) the analysis should assume overflowing semantics for operations on integers and (3) the user should be presented with high-risk security bugs with few false positives.

1.2 Related Work

Bit-precise static analysis originates in hardware model checking, where the bit is the natural unit of information. Work around tools like SMV [13] prompted the research for efficient solvers based on BDDs or SAT techniques. Software model checkers like CBMC [16], SatAbs [17], F-Soft [32] build on efficient SAT solvers to analyze operations on machine integer as Boolean circuits. However, the state explosion problem limits those tools to bounded model checking, where integers are imprecisely modeled using only a few bits. Tools for automatic software testing like SAGE [25] manage to accurately model machine integers as bit-vectors by giving up on (bounded) completeness. All these tools have been reported to find bugs in real C/C++ programs, most of them related to buffer overflows. Bug-finders like PolySpace [43] and Coverity [35] target integer overflow bugs too, based on abstract interpretation, symbolic simulation and SMT solvers.

Interestingly, none of these tools has publicly reported integer overflow bugs, which sustains our claim that these bugs are more subtle than many others. So far, the best attempt at preventing these bugs has been the creation of safe libraries for integer arithmetic operations and casts, like SafeInt [39], intsafe [29] and the CERT Secure integer library [15]. One should keep in mind that it is sometimes subtle to prevent integer overflows, even with these libraries. The best example of this is the allocation of memory through `new` in C++. Figure 4 presents a C++ program which is guaranteed to cause a buffer overflow (with 32-bits integers), despite the test `arr == NULL` that filters failed allocations.

```
void main()
{
  uint siz = 0x40000000;
  int *arr = new int[siz];
  if (arr == NULL) return;
  for (int j = 0; j < siz; ++j) {
    arr[j] = 0;
  }
}
```

```
...
uint siz = 0x40000000;
uint safeint_check =
  siz * SafeInt<uint>(sizeof(int));
uint intsafe_check;
if (FAILED(UIntMult(siz, sizeof(int),
                    &intsafe_check))) ...;
uint cert_check =
  multui(siz, sizeof(int));
int *arr = new int[siz];
...
```

Figure 4: Tricky prevention of integer overflows with `new` in C++

This is because here `sizeof(int) * siz` evaluates to 0! Therefore the allocation returns a pointer to an array of 0 elements. The current version of Microsoft's Visual Studio C++ compiler automatically generates defensive code to detect such integer overflows, but it is not the case for all compilers. It is up to the programmer to check that the multiplication will not overflow, possibly by calling a function of the safe library, like shown in Figure 4, where an integer overflow would raise an exception during the computation of `safeint_check`, `intsafe_check` or `cert_check`.

Recently, a flurry of interest for bit-vector solving has been exhibited in the context of SMT solvers. Some of these are Beaver [9], Boolector [10], Mathsat [11], Spear [1], STP [24], Sword [47] and Z3 [20]. The solvers are compared in anual competitions (`http://www.smtcomp.org`) among SMT solvers. The bit-vector division has received strong attention thanks to the number and quality of solvers that enter.

High integrity software development is especially concerned with integer overflows after an integer overflow run-time error (in Ada) led to the loss of Ariane 5 in 1994. The SPARK approach [3] provides a methodology, a safer language (a subset of Ada), an annotation language and automatic/manual provers to guarantee the absence of run-time errors, including integer overflows. This requires a substantial involvement of the programmer from the design to the coding/proving phase. The application of SMT solvers Yices [23], CVC3 [5] and theorem prover Simplify [22] in the context of SPARK [33] has shown some limitations of the encoding of machine integers as mathematical integers: despite a suitable axiomatization, non-linear arithmetic, division and modulo remain problematic.

Righting software [37] deals with the task of improving the quality of existing mainstream industrial software. The greatest industrial success of this approach to date is the application of the static analysis tools PREfix and ESP to detect buffer overflows in Microsoft code base [28].

1.3 Paper Outline

Section 2 introduces some of the notation and conventions that will be used. It is followed with an overview of the tools PREfix and Z3 in Section 3. Section 3.2.2 provides some background on the bit-vector decision procedures in the context of Z3. The work to integrate the two systems is presented in Section 4. In Section 5, we describe how we encoded the absence of integer overflows as propositions in the theory of bit-vectors so that they can be handled efficiently by Z3. In Section 6, we discuss our treatment of security bugs, false positives and ranking of warnings, that make our tool effective at finding bugs in this particular code base. We report the results of our experiments on the code base in Section 7, in particular the kind of bugs uncovered by our tool so far. We conclude in Section 8 with an examination of the possibilities for improvement.

2 Bit-fiddling Preliminaries

In the following, uncapitalized letters like x stand for bit-vectors while Capitalized Letters like N stand for constant sizes. Unless said otherwise, N is used for arbitrary, but fixed, sized bit-vectors. The zero and one bits are called 0 and 1 respectively. Superscripts like N in 0^N stand for repetition of bits in a bit-vector. We omit the superscript if it is 1. Thus, a single bit can also be used as a bit-vector of length 1. Concatenation of bit-vectors is denoted $\oplus$, which has higher precedence than any other operator. We represent bit-vectors with the most significant bit first, so that integer 1 is represented by N-bits bit-vector $0^{N-1} \oplus 1$. We use brackets to extract single bits, like in $x[0]$. The result is a single bit. Sub-ranges are extracted using two indices, like in $x[1:0]$. Finally, we distinguish equality of bit-vectors $\simeq$ from usual equality. *E.g.*, we always have $x \simeq x[N-1:0]$ and $x[1:0] \simeq x[1] \oplus x[0]$.

The standard arithmetic operations are available in two forms on bit-vectors, signed and unsigned. We use subscripts to distinguish the two forms. For example, $\leq_s$ is signed less-than comparison and $\leq_u$ is unsigned less-than comparison. For $\leq$ and other comparison operations, this simply reflects the fact that signed and unsigned comparisons do not have the same semantics. *E.g.*, the bit-vectors that represent zero and `INT_MIN` are ordered differently when interpreting them as unsigned or signed integers: $0^N <_u 1 \oplus 0^{N-1}$ but $1 \oplus 0^{N-1} <_s 0^N$. We extend this notation to these arithmetic operations that have the same semantics on signed and unsigned integers: addition $+$, subtraction $-$ and multiplication $\times$. Indeed, the resulting bit-vector is the same, only the interpretation of this bit-vector as an integer changes. For example, distinguishing subtraction on unsigned integers $-_u$ from subtraction on signed integers $-_s$ helps us expressing non-overflowing propositions, which depend on the signedness of the subtraction. *E.g.*, for $N > 1$, there is an overflow in the expression $0^N -_u 0^{N-1} \oplus 1$, but not in the expression $0^N -_s 0^{N-1} \oplus 1$. When simply evaluating the result of a bit-vector arithmetic operation, where this distinction is useless, we will omit the subscript.

We use the special equality symbol $\doteq$ to denote a definition, and $ite(x, y, z)$ to denote a conditional expression. When y and z are Booleans, it is equivalent to $(x \wedge y) \vee (\neg x \wedge z)$.

Although it is not prescribed by the C and C++ standards [30, 31], we assume machine integers are encoded in 2-complement notation. (Otherwise, there is not much more to say.)

3 PREfix and Z3

3.1 The PREfix Static Analysis Engine

PREfix [14] is a bug-finding tool based on symbolic simulation, initially developed at Intrinsa between 1994 and 1999, and at Microsoft since 1999. Since 2000, running PREfix analysis on the Microsoft codebase, as well as on many other Microsoft products, has been a requirement for every new release. During development of the new version of the product codebase we analyzed, PREfix reported over 2500 bugs, of which 72% have been fixed. The remaining 28% are false positives, dead code or innocuous bugs.

The architecture of PREfix is that of a static analysis engine, allowing to quickly add new checking capabilities. It traverses the call-graph bottom-up, analyzing one function at a time. The result of this analysis is a set of *outcomes* for each function (a.k.a. a *model*), that comprise each a set of *guards*, *constraints* and *results* [14]. In the terminology of a specification language like JML [38], Spec# [4] or ACSL [6], PREfix generates behaviors for each function, where *guards* are assumes clauses, *constraints* are requires clauses and *results* are ensures clauses. Or equivalently, the formula *guards* $\Rightarrow$*constraints* is part of the function precondition, and the formula *old(guards)* $\wedge$ *b* $\Rightarrow$*results* is part of the function postcondition, where *b* is a fresh Boolean variable expressing the fact that some path through the function (but not all) satisfying to *guards* leads to *results*.

As expected, PREfix only analyzes a subset of the paths through a function, and it models aliasing with unsound heuristics. For each path it analyzes, PREfix maintains three sets of propositions.

- The set of *guards* collects the tests from if-statements and loops, as well as the guards from the selected outcomes of the functions called.
- The set of *constraints* collects checks which are not known to be *true* or *false*.
- The set of *facts* includes the set of guards, together with the equalities implied by assignments.

As it unrolls a path, PREfix creates an SSA equivalent path, whose variables are used in the propositions just mentioned.

At every branching in the function, be it caused by a test or a call, the simulation engine queries the solver to know if the test or the guard (corresponding to an outcome of the function called) is valid (always true), unsatisfiable (always false) or none of these (because both branches are feasible). The solver's three-valued logic answer can be *true*, meaning the query is valid, *false*, meaning the query is unsatisfiable or *don't know* if the query is provably neither valid or unsatisfiable, or if the solver could not determine the correct answer. The simulation engine uses this information to avoid analyzing unfeasible paths.

All safety properties targeted by PREfix are encoded as assertions in the code (possibly involving instrumentation variables) whose violations are considered as bugs. For each one of these *checks*, the simulation engine also queries the solver, with the same three-valued logic answer.

- The answer is *true*: the check is valid on this path.
- The answer is *false*: the check is invalid on this path. PREfix issues a corresponding warning.
- The answer is *don't know*: PREfix adds the check to the set of constraints for this path.

At the end of a path, the sets of *guards*, *constraints* and *facts* are mapped, if possible, to values reachable from the parameters, return value and global variables, and become the *guards*, *constraints* and *results* of a new outcome for the function.

Figure 5 shows an example of path selected by PREfix, consisting in the lines 3, 6, 7, 8 and 11 in function `get_name`. We illustrate now how PREfix analyzes this path, although this simplification does not do justice to the more complex and efficient actual analysis. At line 3, propositions $\texttt{init(buf}_0\texttt{)}$ and $\texttt{init(size}_0\texttt{)}$ are added to the set of facts, to denote that parameters are necessarily initialized. At line 6, propositions $\texttt{init(status}_0\texttt{)}$ and $\texttt{status}_0\ \texttt{==}\ \texttt{0}$ are added to the set of facts. At line 7, assuming outcome init_name_0 is chosen here, the solver cannot determine whether $\texttt{size}_0\ \texttt{==}\ \texttt{0}$ is valid or unsatisfiable. Therefore the corresponding proposition is added to the set of guards and result $\texttt{init(status}_1\texttt{)}$ and $\texttt{status}_1\ \texttt{==}\ \texttt{0}$ are added to the set of facts. The solver determines that test $\texttt{!}\ \texttt{status}_1\ \texttt{>=}\ \texttt{0}$ at line 8 is false on this path, therefore the simulation engine proceeds with the (empty) else-branch. At line 11, upon reading the value of variable `name`, PREfix issues a query for check $\texttt{init(name}_0\texttt{)}$, which the solver is able to determine is false. So the simulation engine issues a warning that function `get_name` is *using uninitialized memory* `name` on the path analyzed.

A very carefully crafted part of PREfix is the ranking of warnings that allows discarding false positives. PREfix keeps metrics on each warning issued, so that these metrics can be used later on to compute a score. This way, a user needs only look at the warnings of lowest score, which should be of higher quality, *i.e.*, with fewer false positives.

Overall, PREfix static analysis engine is both modular and interprocedural, thanks to its syntax-driven splitting of paths.

```
1  NTSTATUS init_name(char **outname, uint n);
2
3  NTSTATUS get_name(char* buf, uint size)
4  {
5    char* name;
6    NTSTATUS status = STATUS_SUCCESS;
7    status = init_name(&name, size);
8    if (! NT_SUCCESS(status)) {
9      goto error;
10   }
11   strcpy(buf, name);
12 error:
13   return status;
14 }
```

model for function `init_name`

outcome init_name_0:
  guards: `n == 0`
  results: `result == 0`

outcome init_name_1:
  guards: `n > 0`; `n <= 65535`
  results: `result == 0xC0000095`

outcome init_name_2:
  guards: `n > 0`; `n <= 65535`
  constraints: valid(`outname`)
  results: `result == 0`; init(`*outname`)

Figure 5: Path selection in PREfix

3.2 The SMT Solver Z3

Z3 [20] is a state-of-the-art Satisfiability Modulo Theories (SMT) solver. An SMT solver is a theorem prover that can establish theorems for formulas that combine one or more *theories*, where a theory is given by a sets of sorts, functions over those sorts and axioms about these functions. One such theory is the theory of bit-vectors; other theories commonly found in SMT solvers are the theories of real arithmetic and linear integer arithmetic, the theory of arrays à la McCarthy [41], and the theory of tuples and algebraic datatypes [44]. All of these theories are decidable in the quantifier-free case, that is, there are special purpose algorithms, also known as decision procedures, that decide satisfiability of constraints over each of the theories. To decide the validity of a formula, one typically shows the unsatisfiability of the negated formula. Formulas are free to mix functions defined from one of the respective theories; decidability is preserved. Z3 integrates a modern DPLL-based SAT solver, a core theory solver that handles ground equalities over uninterpreted functions, and *satellite solvers* for the theories. PREfix requires almost exclusively the theory of bit-vectors. We currently encode constraints over floating points using a rough approximation with arithmetic over reals.

Currently, Z3 is used in several projects related to Microsoft Research, including Spec#/Boogie [4, 21], Pex [45], HAVOC [36], Vigilante [18], a verifying C compiler (VCC), SLAM/SDV [2], SAGE [42], and Yogi [27].

3.2.1 Interfacing with Z3

Users interface with Z3 using one of the available APIs. Z3 supports three different text formats and exposes programmatic APIs for C, OCaml, and .NET. Our integration with PREfix uses the C-based API. The main functionality exposed by the programmatic APIs is to build expressions, assert expressions of Boolean type, and check for consistency of the asserted expressions. It is also possible to *push* and *pop* local scope such that assertions inside a *push* can be retracted after a matching *pop*. A sample C program using the API is shown in Figure 6. It checks that both $a \leq_u a \times b$ and $\neg(a \leq_u a \times b)$ are satisfiable.

3.2.2 Bit-precise Reasoning in Z3

The implementation of bit-level reasoning in the current version of Z3 uses a preprocessor based on rewriting rules followed by an eager reduction of bit-level constraints into propositional constraints.

```
Z3_sort* bv32 = Z3_mk_bv_sort(ctx, 32);           // create the sort bv32
Z3_symbol a_s = Z3_mk_string_symbol(ctx, "a");    // create the name 'a'
Z3_ast* a = Z3_mk_const(ctx, a_s, bv32);          // create a 32-bit constant 'a'
Z3_symbol b_s = Z3_mk_string_symbol(ctx, "b");    // create the name 'b'
Z3_ast* b = Z3_mk_const(ctx, b_s, bv32);          // create a 32-bit constant 'b'
Z3_ast* ab = Z3_mk_bvmul(ctx, a, b);              // create a * b
Z3_ast* e = Z3_mk_bvule(ctx, a, ab);              // create e := a <= a * b
Z3_push(ctx);                                     // push a scope
Z3_assert_cnstr(ctx, e);                          // assert e
Z3_lbool result1 = Z3_check(ctx);                 // 'e' is sat, result1 = Z3_true
Z3_pop(ctx, 1);                                   // pop one scope level
Z3_push(ctx);                                     // push a scope
Z3_ast* ne = Z3_mk_not(ctx, e);                   // create ne := !(a <= a * b)
Z3_assert_cnstr(ctx, ne);                         // assert ne
Z3_lbool result2 = Z3_check(ctx);                 // 'ne' is sat, result2 = Z3_true
Z3_pop(ctx, 1);                                   // pop one scope level
```

Figure 6: Sample Z3 API usage

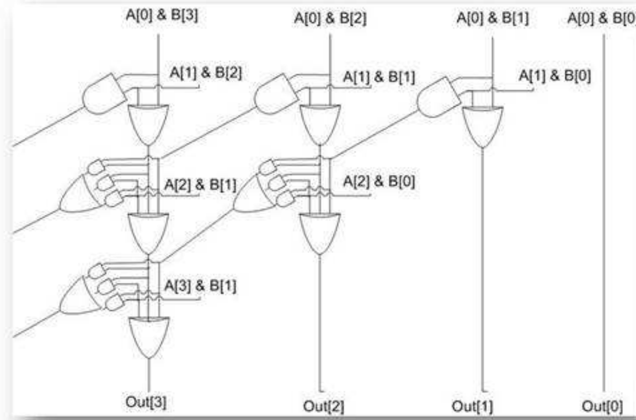


Figure 7: A circuit for multiplying two 4-bit vectors

This reduction is called *bit-blasting*. The propositional constraints are then handled by the SAT solver component. We illustrate the encoding of bit-level multiplication for two vectors of length 4 in Figure 7.

Each gate represents one or more propositional connectives. Z3 associates with each gate in the multiplication circuit one or more propositional subformulas. For example, the expression $a[3:0] \times b[3:0]$ is represented by four bit-level expressions we will call $z_{0,0}$, $z_{1,1}$, $z_{2,2}$, $z_{3,3}$, where:

$$z_{0,0} \doteq a[0] \wedge b[0] \qquad z_{1,1}, u_{1,1} \doteq ha(a[0] \wedge b[1], a[1] \wedge b[0])$$

$$\begin{aligned} z_{2,1}, u_{2,1} &\doteq ha(a[0] \wedge b[2], a[1] \wedge b[1]) \\ z_{2,2}, u_{2,2} &\doteq fa(u_{1,1}, z_{2,1}, a[2] \wedge b[0]) \end{aligned} \qquad \begin{aligned} z_{3,1}, u_{3,1} &\doteq ha(a[0] \wedge b[3], a[1] \wedge b[2]) \\ z_{3,2}, u_{3,2} &\doteq fa(u_{2,1}, z_{3,1}, a[2] \wedge b[1]) \\ z_{3,3}, u_{3,3} &\doteq fa(u_{2,2}, z_{3,2}, a[3] \wedge b[0]) \end{aligned}$$

and $z, u \doteq ha(x, y) \doteq (x \otimes y, x \wedge y)$ assigns to z, u the result of a half-adder, and $z, u \doteq fa(x, y, w) \doteq$

$(x \otimes y \otimes w, (x \wedge y) \vee (x \wedge w) \vee (y \wedge w))$ assigns to z, u the result of a full-adder. In Figure 7, z variables are represented by vertical results and u variables by oblique carries.

If $a \times b$ is used in an atom of the form $a \leq_u a \times b$, then the atom is equivalent to the formula:

$$(z_{3,3} \wedge \neg a[3]) \vee ((z_{3,3} \vee \neg a[3]) \wedge ((z_{2,2} \wedge \neg a[2]) \vee ((z_{2,2} \vee \neg a[2]) \wedge ((z_{1,1} \wedge \neg a[1]) \vee ((z_{1,1} \vee \neg a[1]) \wedge (z_{0,0} \vee \neg a[0]))))) \quad (1)$$

where $z_{0,0}, z_{1,1}, z_{2,2}$, and $z_{3,3}$ are shorthands for the formulas defined using the gates.

Propositional formulas are finally clausified using a Tseitsin [46] style conversion algorithm. The algorithm replaces nested Boolean connectives by fresh propositional atoms and adds clauses that constrain the fresh atoms to encode the connectives. Terms in Z3 use maximal structure sharing, such that common sub-expressions are represented using a unique node. Then, common sub-expressions are clausified only once, so if the CNF conversion encounteres a sub-formula that has already been clausified, it produces the fresh predicate that was produced as a result of the first time clausification was invoked on the sub-expression.

We illustrate clausification using formula (1). In a top-down traversal of the formula, we introduce fresh atoms $p_1, p_2, \ldots$ for every unique sub-formula. Clauses for the top four subformulas are listed below.

$$(p_1 \Rightarrow p_2 \vee p_3) \wedge (p_2 \Rightarrow p_1) \wedge (p_3 \Rightarrow p_1) \wedge \qquad \textit{i.e.}, p_1 \doteq \underbrace{(z_{3,3} \wedge \neg a[3])}_{p_2} \vee \underbrace{(\ldots \wedge \ldots)}_{p_3}$$

$$(p_2 \Rightarrow z_{3,3}) \wedge (p_2 \Rightarrow \neg a[3]) \wedge (a[3] \vee \neg z_{3,3} \vee p_2) \wedge \qquad \textit{i.e.}, p_2 \doteq z_{3,3} \wedge \neg a[3]$$

$$\begin{array}{l}(z_{3,3} \Rightarrow u_{2,2} \vee z_{3,2} \vee p_4) \wedge (z_{3,3} \wedge u_{2,2} \wedge z_{3,2} \Rightarrow p_4) \wedge \\ (z_{3,3} \wedge u_{2,2} \wedge p_4 \Rightarrow z_{3,2}) \wedge (z_{3,3} \wedge z_{3,2} \wedge p_4 \Rightarrow u_{2,2}) \wedge \\ (p_4 \Rightarrow z_{3,3} \vee z_{3,2} \vee u_{2,2}) \wedge (z_{3,2} \Rightarrow z_{3,3} \vee p_4 \vee u_{2,2}) \wedge \\ (u_{2,2} \Rightarrow z_{3,3} \vee z_{3,2} \vee p_4) \wedge (p_4 \wedge z_{3,2} \wedge u_{2,2} \Rightarrow z_{3,3}) \wedge \end{array} \qquad \textit{i.e.}, z_{3,3} \doteq u_{2,2} \otimes z_{3,2} \otimes \underbrace{(a[3] \wedge b[0])}_{p_4}$$

$$(p_3 \Rightarrow p_5) \wedge (p_3 \Rightarrow p_6) \wedge (p_5 \wedge p_6 \Rightarrow p_3) \qquad \textit{i.e.}, p_3 \doteq (\underbrace{(z_{3,3} \vee \neg a[3])}_{p_5} \wedge \underbrace{\ldots}_{p_6})$$

The conversion into clausal form performs some straight-forward optimizations. In particular, when building a multiplication circuit for an expression of the form $1001 \times b$, where one argument is a fixed constant (the binary representation of the numeral 9), then the circuit representation can be simplified by performing standard algebraic manipulations, such as $x \otimes y \otimes 1$ being simplified to $x \Leftrightarrow y$. The number of clauses that results from a simplified circuit is consequently drastically reduced.

Unfortunately, as the construction suggests, the general representation of multiplication and division circuits require $O(N^2)$ clauses and fresh atoms. The quest for efficient techniques, or techniques that work well on applications, for solving bit-vector multiplication (and division) constraints is therefore an important, but unsettled research area [12].

3.2.3 Bit-precise Reasoning in Practice

Modern SMT solvers use a highly optimized SAT solver core. The SAT solver core is capable of propagating constraints among several thousand clauses in milliseconds using efficient indexing techniques. Nevertheless, the scale issues with handling multiplication constraints are easily observed. Figure 8 shows the time, number of literals, and number of clauses that are created from simple circuits for multiplying two N-bit numbers. In the figure, N ranges from 1 to 64 bits.

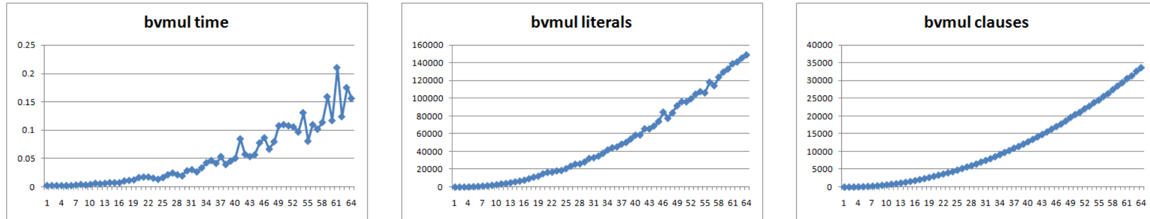


Figure 8: Saturation time/number of literals/number of initial and generated clauses for multiplication circuits

4 Integration

The integration of PREfix and Z3 consisted in replacing the calls in PREfix simulation engine to PREfix solver by equivalent calls to Z3's C API. To get the same three-valued logic answers, we first query Z3 for the satisfiability of the negated queried predicate. If the answer is *unsatisfiable*, then we return *true*. Otherwise we query Z3 for the satisfiability of the queried predicate. If the answer is *unsatisfiable*, then we return *false*. Otherwise we return *don't know*.

To express these queries in Z3, it is necessary to translate PREfix propositions into Z3 propositions. This involves translating PREfix SSA variables and constants of machine integer type to Z3 terms of the appropriate bit-vector type (meaning the number of bits). Operations on machine integers translate to operations on bit-vectors. We also translate pointer values into bit-vectors, which is sound, and floating-point values into reals, which is the only unsound part of the translation.

A not-so-easy part of the integration consisted in retrieving all the information maintained by PREfix about a path into propositions. Indeed, the actual behavior of the simulation engine is not as simple as the one presented in Section 3.1. First, information about the value of variables, even partial, is not stored as propositions in the set of facts, but rather directly in data-structures pointed to by the variable. Secondly, the simulation engine encodes some of the properties into integer variables (*e.g.*, the validity of pointers) and others in *ad. hoc.* ways (*e.g.*, being initialized). As a starting point, we translated propositions added to the set of PREfix facts into Z3 propositions and asserted them in the context maintained by Z3 along the way. We solved the first problem by forcing the generation of Z3 propositions in those cases too. We solved the second problem by keeping the parts of PREfix solver which treat these special cases.

A strong requirement for our integration was to generate only ground propositions in Z3, without quantified axioms, so that the queries can be answered without going through the expensive process of matching and quantifier instantiation. This allowed us to keep the timeout for each query to a very low 100ms.

PREfix maintained only atomic propositions in its set of facts, in order to simplify the task of its solver. Although the current integration follows this restriction, a promising oportunity is to handle some disjunctions directly to Z3, instead of systematically splitting paths inside the simulation engine. As a proof-of-concept implementation, we now handle class casts in C++ as disjunctions in Z3.

Another oportunity for efficiency gains is the rollback mechanism offered by Z3. We started supporting such a mechanism at the level of the simulation engine for backtracking from failed attempts at simulating an outcome for a call, without having to discard the path like previously.

During these experiments, where fine-tuning made a great difference, it was essential to have among the authors of the paper one owner of each tool PREfix and Z3.

5 Integer Overflow Checks

For each operation that can overflow, whether it is arithmetic or a cast, we devise an appropriate *check*, *i.e.*, a proposition in the theory of bit-vectors which expresses that the operation does not overflow. These checks are slighly different from traditional assertions, in that they cannot be easily expressed in the source language C/C++, and, as mentioned in Section 1, they do not constrain the value of program variables. The latter ensures that we analyze programs with an overflowing semantics. The former may be surprising, as safe libraries, like SafeInt [39], intsafe [29] and the CERT Secure integer library [15], do express similar checks as assertions in C/C++. Although we could choose these expressions for our propositions, this would be very inefficient. Our propositions are more efficient, at the cost of not being expressible in C/C++.

Although we collectively designate them as integer overflows, we distinguish overflows proper, where the mathematical result is larger than the value represented, from underflows, where the mathematical result is smaller than the value represented. We denote the non-overflowing predicate $[\![.]\!]_+$ and the non-underflowing predicate $[\![.]\!]_-$. They both take as term argument an operation, and return a first-order logic proposition in the theory of bit-vectors, which expresses respectively the absence of overflow or underflow during the operation. *E.g.*, returning to the difference between signed and unsigned subtraction seen in Section 2, the following are valid formulas for $N > 1$:

$$[\![0^N -_u 0^{N-1} \oplus 1]\!]_+ \simeq \mathit{true} \qquad [\![0^N -_s 0^{N-1} \oplus 1]\!]_+ \simeq \mathit{true}$$
$$[\![0^N -_u 0^{N-1} \oplus 1]\!]_- \simeq \mathit{false} \qquad [\![0^N -_s 0^{N-1} \oplus 1]\!]_- \simeq \mathit{true}$$

(Actually, the last formula is not true for $N = 1$. Our non-overflowing predicates are correct for any N, including $N = 1$.)

5.1 Arithmetic Operations

There are four binary arithmetic operations (addition $+$, subtraction $-$, multiplication $\times$ and division $/$) and one unary arithmetic operation (negation $-$). Remainder operation, which can neither overflow nor underflow, is not considered. (Signed remainder follows sign of dividend.) Bit-vector operands of each binary arithmetic operation should have the same size N.

Addition

Unsigned addition can only overflow. To check the absence of overflow, we extend bit-vector operands x and y by one zero-bit, perform the addition on the extended bit-vectors, and check that the result fits in N bits.

$$[\![x +_u y]\!]_+ \doteq (0 \oplus x + 0 \oplus y)[N] \simeq 0$$

Signed addition can both overflow and underflow. There are three cases, depending on the sign of x and y: (1) if x and y have opposite signs, the addition can neither overflow nor underflow; (2) if x and y are both non-negative, the addition can only overflow, which necessarily leads to a negative result; (3) conversely, if x and y are both negative, the addition can only underflow, which necessarily leads to a non-negative result.

$$[\![x +_s y]\!]_+ \doteq x[N-1] \simeq 0 \wedge y[N-1] \simeq 0 \Rightarrow (x+y)[N-1] \simeq 0$$

$$[\![x +_s y]\!]_- \doteq x[N-1] \simeq 1 \wedge y[N-1] \simeq 1 \Rightarrow (x+y)[N-1] \simeq 1$$

Subtraction

Unsigned subtraction can only underflow. Checking that the subtracted operand y is smaller than the other operand x is a necessary and sufficient condition for non-underflowing.

$$[\![x -_u y]\!]_- \doteq y \leq_u x$$

Signed subtraction can both overflow and underflow. There are three cases, depending on the sign of x and y: (1) if x and y have the same sign, the subtraction can neither overflow nor underflow; (2) if x is non-negative and y is negative, the subtraction can only overflow, which necessary leads to a negative result; (3) conversely, if x is negative and y is non-negative, the addition subtraction can only underflow, which necessary leads to a non-negative result.

$$[\![x -_s y]\!]_+ \doteq x[N-1] \simeq 0 \wedge y[N-1] \simeq 1 \Rightarrow (x-y)[N-1] \simeq 0$$

$$[\![x -_s y]\!]_- \doteq x[N-1] \simeq 1 \wedge y[N-1] \simeq 0 \Rightarrow (x-y)[N-1] \simeq 1$$

Multiplication

Unsigned multiplication can only overflow. To check the absence of overflow, we extend bit-vector operands x and y by N zero-bits, perform the multiplication on the extended bit-vectors, and check that the result fits in N bits.

$$[\![x \times_u y]\!]_+ \doteq (0^N \oplus x \times 0^N \oplus y)[2 \times N - 1 : N] \simeq 0^N$$

Signed multiplication can both overflow and underflow. There are two cases, depending on the sign of x and y: (1) if x and y have the same sign, the multiplication can only overflow; (2) conversely, if x and y are opposite signs, the multiplication can only underflow. To check respectively the absence of overflow or underflow, we sign-extend bit-vector operands x and y by N bits, perform the multiplication on the extended bit-vectors, and check that the result fits in N bits.

$$[\![x \times_s y]\!]_+ \doteq x[N-1] \simeq y[N-1] \Rightarrow \\ (x[N-1]^N \oplus x \times y[N-1]^N \oplus y)[2 \times N - 1 : N-1] \simeq 0^{N+1}$$

$$[\![x \times_s y]\!]_- \doteq x[N-1] \not\simeq y[N-1] \Rightarrow \\ (x[N-1]^N \oplus x \times y[N-1]^N \oplus y)[2 \times N - 1 : N-1] \simeq 1^{N+1}$$

Division

Unsigned division cannot overflow or underflow. Signed division can only overflow, when performed over the minimal integer value represented by bit-vector $1 \oplus 0^{N-1}$ and -1 represented by bit-vector 1^N. This is because there is one less positive numbers than there are negative numbers. To check the absence of overflow, we exclude this case.

$$[\![x /_s y]\!]_+ \doteq \neg(x \simeq 1 \oplus 0^{N-1} \wedge y \simeq 1^N)$$

Negation

Like signed division, negation can only overflow, for the same reason. To check the absence of overflow, we exclude this case.

$$[\![-_s\, x]\!]_+ \;\doteq\; \neg(x \simeq 1 \oplus 0^{N-1})$$

5.2 Cast Operations

A cast operation converts a bit-vector operand x over N bits, seen as signed or unsigned, to an M-bits bit-vector, also seen as signed or unsigned. We denote it as $(N_{s/u} \to M_{s/u})\, x$. Therefore, there are as many casts as there are ordered pairs of different integer types. With 6 integer types on 32-bits machines, there are 30 different cast operations. With 8 integer types on 64-bits machines, there are 56 different cast operations.

Unsigned Casts

A cast from an unsigned integer type to another unsigned integer type cannot underflow, and it can overflow only when $M < N$. In this case, checking the absence of overflow amounts to checking that x fits in M bits.

$$[\![(N_u \to M_u)\, x]\!]_+ \;\doteq\; x[N-1:M] \simeq 0^{N-M}$$

Casts From Unsigned to Signed

A cast from an unsigned integer type to a signed integer type cannot underflow, and it can overflow only when $M \leq N$. In this case, checking the absence of overflow amounts to checking that x fits in $M-1$ bits (because the most significant bit should be 0).

$$[\![(N_u \to M_s)\, x]\!]_+ \;\doteq\; x[N-1:M-1] \simeq 0^{N-M+1}$$

Signed Casts

A cast from a signed integer type to another signed integer type can both overflow or underflow only when $M < N$. It can only overflow when x is non-negative, in which case we must check that x fits in $M-1$ bits (because the most significant bit should be 0). It can only underflow when x is negative, in which case we must check that x fits in $M-1$ bits (because the most significant bit should be 1).

$$[\![(N_s \to M_s)\, x]\!]_+ \;\doteq\; x[N-1] \simeq 0 \Rightarrow x[N-1:M-1] \simeq 0^{N-M+1}$$

$$[\![(N_s \to M_s)\, x]\!]_- \;\doteq\; x[N-1] \simeq 1 \Rightarrow x[N-1:M-1] \simeq 1^{N-M+1}$$

Casts From Signed to Unsigned

A cast from a signed integer type to an unsigned integer type can overflow only when $M < N-1$ and it can underflow for any values of N and M (even 1). It can only overflow when x is non-negative, in which case we must check that x fits in M bits. It does underflow whenever x is negative.

$$[\![(N_s \to M_u)\, x]\!]_+ \;\doteq\; x[N-1] \simeq 0 \Rightarrow x[N-1:M] \simeq 0^{N-M}$$

$$[\![(N_s \to M_u)\, x]\!]_- \doteq x[N-1] \simeq 1$$

5.3 Sound Approximations for Multiplication Overflows

Based on initial experiments, we quickly noted that the non-overflowing checks for multiplication are too costly in practice, leading to many timeouts of Z3. (We set a very low timeout of 100ms.) As shown in Section 5.1, the initial non-overflowing check we used was based on the most straight-forward approach that consists in computing a result on $2 \cdot N$ bits and checking if any of the leading N bits is set. Instead, we implemented in Z3 a far more efficient approach that does not require to compute the leading $N-1$ bits of the result. It is based on the efficient overflow checks by Gök *et al.* [26]. This approach still requires to compute the lowest $N+1$ bits of the result, so we devised sound approximate checks that run even faster, at the cost of being incomplete. In this Section, we will investigate how the naïve overflow checks compare to the more efficient version as well as the approximate version.

Given a costly initial check $[\![?]\!]_\pm$, our goal is to come up with propositions ϕ and ψ that are cheaper to check in Z3, such that ϕ is a stronger proposition than $[\![?]\!]_\pm$, and ψ is a weaker proposition.

$$\phi \Rightarrow [\![?]\!]_\pm \Rightarrow \psi$$

Given such propositions ϕ and ψ, checking the validity of $[\![?]\!]_\pm$ can be answered by checking the validity of the stronger ϕ, and checking the unsatisfiability of $[\![?]\!]_\pm$ can be answered by checking the unsatisfiability of the weaker ψ.

5.3.1 Unsigned Multiplication

Looking at unsigned multiplication first, we note that

$$y \leq 2^{N-\lfloor \log_2(x) \rfloor - 1} \Rightarrow [\![x \times_u y]\!]_+ \Rightarrow y < 2^{N-\lfloor \log_2(x) \rfloor}.$$

Given that $\lfloor \log_2(x) \rfloor$ is the position of the most significant 1-bit of x, we can rewrite propositions ϕ and ψ. ψ rewrites to a cascading if-then-else proposition, with the strict inequality translated as expected zeros.

$$\begin{aligned} y < 2^{N-\lfloor \log_2(x) \rfloor} \quad = \quad & ite(x[N-1] \simeq 1, y[N-1:1] \simeq 0^{N-1}, \\ & \quad ite(x[N-2] \simeq 1, y[N-1:2] \simeq 0^{N-2}, \\ & \quad \quad \dots \\ & \quad \quad \quad ite(x[0] \simeq 1, true, true) \dots)) \end{aligned} \tag{2}$$

ϕ also rewrites to a cascading if-then-else proposition, with the non-strict inequality translated as equality with the bound or strict inequality, like above.

$$\begin{aligned} y \leq 2^{N-\lfloor \log_2(x) \rfloor - 1} \quad = \quad & ite(x[N-1] \simeq 1, y \simeq 0^{N-1} \oplus 1 \vee y[N-1:0] \simeq 0^N, \\ & \quad ite(x[N-2] \simeq 1, y \simeq 0^{N-2} \oplus 1 \oplus 0^1 \vee y[N-1:1] \simeq 0^{N-1}, \\ & \quad \quad \dots \\ & \quad \quad \quad ite(x[0] \simeq 1, true, true) \dots)) \end{aligned} \tag{3}$$

When N is a power of 2, we can build families of propositions $(\phi)_k$ and $(\psi)_k$, such that

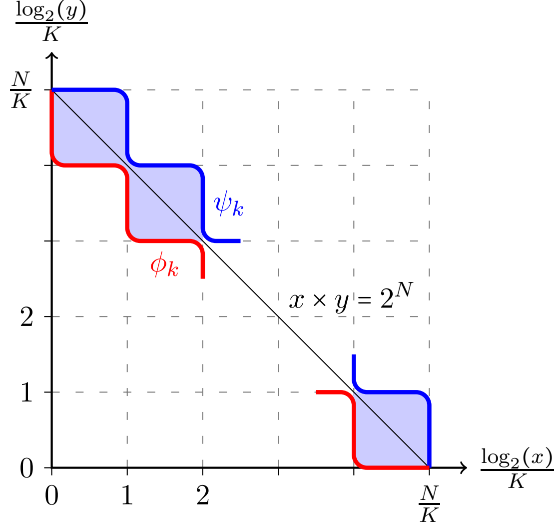


Figure 9: Approximation (ϕ_k, ψ_k) of the non-overflowing multiplication check

$$\phi_{\log_2(N)} \Rightarrow \ldots \Rightarrow \phi_1 \Rightarrow \phi_0 \simeq y \leq 2^{N-\lfloor \log_2(x) \rfloor - 1}$$

$$y < 2^{N-\lfloor \log_2(x) \rfloor} = \psi_0 \Rightarrow \psi_1 \ldots \Rightarrow \psi_{\log_2(N)}$$

The definitions of ψ_k and ϕ_k are respectively generalizations of formulas (2) and (3). We use the shorthand K for 2^k.

$$\begin{aligned}
\psi_k \quad \doteq \quad & ite(x[N-1:N-K] \not\simeq 0^K, y[N-1:K] \simeq 0^{N-K}, \\
& \quad ite(x[N-K-1:N-2\times K] \not\simeq 0^K, y[N-1:2\times K] \simeq 0^{N-2\times K}, \\
& \quad\quad \ldots \\
& \quad\quad\quad ite(x[K-1:0] \not\simeq 0^K, true, true)\ldots))
\end{aligned}$$

$$\begin{aligned}
&\phi_k \ \doteq \\
&\quad ite(x[N-1:N-K] \not\simeq 0^K, y \simeq 0^{N-1} \oplus 1 \vee y[N-1:0] \simeq 0^N, \\
&\quad\quad ite(x[N-K-1:N-2\times K] \not\simeq 0^K, y \simeq 0^{N-K-1} \oplus 1 \oplus 0^K \vee y[N-1:K] \simeq 0^{N-K}, \\
&\quad\quad\quad \ldots \\
&\quad\quad\quad\quad ite(x[K-1:0] \not\simeq 0^K, y \simeq 0^{K-1} \oplus 1 \oplus 0^{N-K} \vee y[N-1:2\times N-K] \simeq 0^K, true)\ldots))
\end{aligned}$$

Figure 9 plots pairs of operands (x, y) on a logarithmic scale. Each pair of functions (ϕ_k, ψ_k) defines a step-wise approximation of function $x \times y = 2^N$ from both sides. First, the number $\mathcal{N}_\times$ of pairs of operands which do not lead to an overflow is

$$\mathcal{N}_\times = \sum_{x=0}^{2^N-1} \#\left\{y | x \times y < 2^N\right\} = 2^N - 1 + \sum_{x=1}^{2^N-1} \left\lfloor \frac{2^N-1}{x} \right\rfloor.$$

The summation can be bounded as follows:

$$\sum_{x=1}^{2^N-1} \left\lfloor \frac{2^N-1}{x} \right\rfloor < 2^N - 1 + \int_1^{2^N-1} \frac{2^N-1}{x}.dx = (2^N-1) \times \left(1 + \ln(2^N-1)\right) \approx \frac{2^N \times (N+1)}{\log_2(\mathrm{e})}$$

$$\sum_{x=1}^{2^N-1} \left\lfloor \frac{2^N-1}{x} \right\rfloor > -2^N + 1 + \int_0^{2^N-1} \frac{2^N-1}{x+1}.dx = (2^N-1) \times \left(-1 + \ln(2^N)\right) \approx \frac{2^N \times (N-1)}{\log_2(\mathrm{e})}$$

This leads to approximate bounds for $\mathcal{N}_\times$:

$$\frac{2^N \times N}{\log_2(\mathrm{e})} \lesssim \mathcal{N}_\times \lesssim \frac{2^N \times (N+2)}{\log_2(\mathrm{e})}.$$

Now, the number of pairs of operands $\mathcal{N}_\phi$ that are correctly classified as non-overflowing by ϕ_k is

$$\mathcal{N}_\phi = \sum_{X=0}^{\frac{N}{K}-1} (2^{K\times(X+1)} - 2^{K\times X}) \times 2^{N-K\times(X+1)} = \frac{N}{K} \times \left(2^N - 2^{N-K}\right)$$

The last summation goes from $\mathcal{N}_\phi = 2^N - 1$ (almost no pairs) for $K = N$ to $\mathcal{N}_\phi = N \times 2^{N-1}$ (of the same order as $\frac{\mathcal{N}_\times}{2}$) for $K = 1$.

Likewise, the number of pairs of operands $\mathcal{N}_\psi$ that are not classified as overflowing by ψ_k is

$$\mathcal{N}_\psi = \sum_{X=0}^{\frac{N}{K}-1} (2^{K\times(X+1)} - 2^{K\times X}) \times 2^{N-K\times X} = \frac{N}{K} \times \left(2^{N+K} - 2^N\right)$$

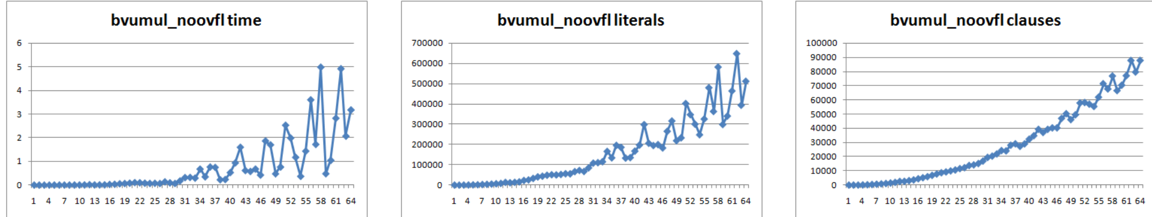
The last summation goes from $\mathcal{N}_\psi = 2^{2\times N} - 2^N$ (almost all pairs) for $K = N$ to $\mathcal{N}_\psi = N \times 2^N$ (of the same order as $\mathcal{N}_\times$) for $K = 1$.

In particular, the most precise pair of approximation functions (ϕ_0, ψ_0) defined by formulas (2) and (3) succeeds in correctly classifying a non-overflowing pair of operands roughly half of the time, but more importantly, it almost always correctly classifies an overflowing pair of operands, with a probability of $1 - \frac{1}{2^N}$. Since we are looking for bugs, this is the side of the approximation we care most about. Our experiments also show that, contrary to the off-by-one bugs that are common among integer overflows related to addition and subtraction, integer overflows related to multiplication do not tend to show only at the fringe, which validates our approach. Notice that ϕ_0 is the same as the preliminary overflow flag, V'_u, from Gök *et al.* [26], for which they give an efficient implementation.

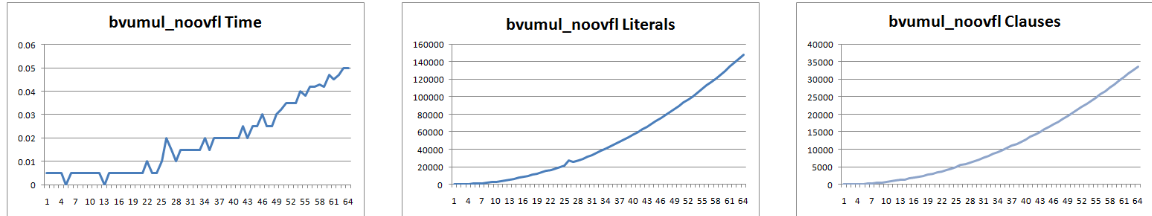
Figure 10 shows the time, number of literals, and number of clauses that are created from the circuits for (1) checking for non-overflows of two N-bit numbers as presented in Section 5, (2) checking for non-overflows using the circuit construction described in [26], and (3) approximate checking of N-bit multiplication using formula (3). In the figure, N ranges from 1 to 64 bits. We observe that the overhead of checking unsigned multiplication overflow can very quickly be dominant. The approximate multiplication overflow checks require both linear time/space overhead and is an order of magnitude faster.

5.3.2 Signed Multiplication

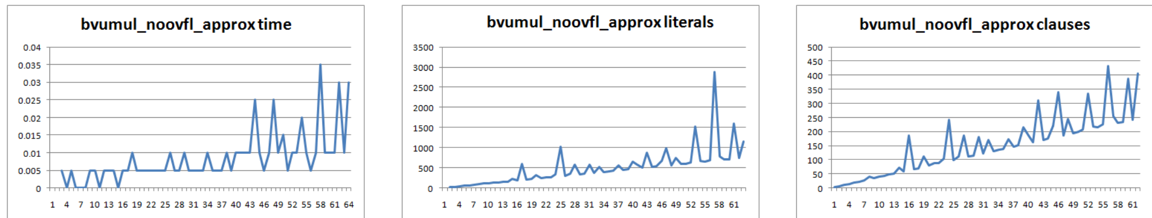
We check for approximate overflows and underflows of signed multiplication by using a reduction to unsigned multiplication. The reduction requires to negate negative numbers. There is a special case when either of the multiplicands is the minimal negative integer value, for which direct negation underflows, so that this case needs to be treated separately. Overall, the same kind of approximations as above applies.



Naïve multiplication overflow detection using $2 \cdot n$ bit-vector multiplication



Optimized $(n + 1)$-bit multiplication based overflow detection



Approximate multiplication overflow detection

Figure 10: Saturation time/number of literals/number of initial and generated clauses for unsigned multiplication overflow circuits, and circuits approximating safe multiplication non-overflow

6 Practical Bug-finding

Finding integer overflows in programs, as defined in Section 5, is utterly useless by itself. The vast majority of potentially overflowing operations, as defined by a straightforward implemention, do not correspond to bugs. The following three developments of our checker were crucial in making bug-finding successful: (1) we added checks for uses of overflown values which can lead to security issues; (2) we identified and discarded broad categories of false positives and (3) we devised a ranking scheme so as to present the user with the most serious warnings first.

6.1 Security Checks for Using an Overflown Value

Based on the security vulnerabilities reported so far related to integer overflows, there are mainly two cases where an integer overflow can lead to a buffer overflow (which is the main security issue with integer overflows.) We do not distinguish overflows proper from underflows here.

- An integer passed as size argument to an an allocation function, such as `malloc` or `calloc`, is

```
S* get_elems(
  uint num)
{
  uint siz = num * sizeof(S);
  S* tab = malloc(siz);
  return tab;
}
```

```
void load_data(
  S* data, uint pos)
{
  uint spos = pos - Gpos;
  data[spos] = get();
}
```

Figure 11: Security integer overflows: allocation size and pointer offset

the result of an integer overflow.

- An integer used as offset in pointer arithmetic or pointer indexing is the result of an integer overflow.

Figure 11 shows an example of each vulnerability. In function `get_elems`, the computation of `siz` could overflow, in which case the array allocated will be smaller than expected. Since the caller has no reason to believe the array is not of the appropriate size, it will almost surely access it beyond its bounds. In function `load_data`, the computation of `spos` could overflow (well, underflow here), in which case array `data` is written beyond its bounds.

Our mechanism for detecting such dangerous uses of overflown values keeps track of whether a value is the result of an integer overflow, and if so records the operation which computed this value. Like tainting, the property of possibly being the result of an integer overflow propagates transitively from the operands to the result of each operation. When a value is used as an allocation size or a pointer offset, we first check whether this value has been flagged as possibly overflown. If so, we build a proposition that expresses that none of the (possibly overflowing) operations leading to this value can be overflowing. Finally, we query Z3 for the validity of this proposition in the current context, and we issue a warning whenever the result is not *true*. Figure 12 shows three examples where Z3 answers respectively *false*, *true* and *don't know*.

In the case of function `ex_false`, `siz` is trivially always the result of an integer overflow and we issue a warning.

In the case of function `ex_true`, `siz` may be the result of an integer overflow, but the test `siz < num` filters all cases where an overflow occurs, therefore the allocation is safe. By querying the validity of the non-overflowing proposition at the point where `tab` is allocated, we are able to identify that there is no problem in this function.

In the case of function `ex_dont_know`, `siz` may be the result of an integer overflow, but we cannot be sure of this without knowing the possible values of parameter `num`. As explained in Section 3.1, the usual way this is handled in PREfix is to add the corresponding proposition to the constraints of the current path. Then, it gets mapped back, if possible, to function parameters in order to generate a constraint that propagates to the callers of the current function. In this process, a warning is issued only if a function always triggers the integer overflow on some path. While system models help generating such paths in the case of buffer overflows, this is not so easy with integer overflows, which makes it likely to miss errors. This is why we chose instead to report these cases as warnings right-away, due to the criticity of the associated bugs. Notice that we assume here that `sizeof(S)` is greater than 1. Had it been equal to 1, or had the allocation size been simply parameter `num`, the corresponding value would not be flagged as possibly overflown, and we would not issue a warning. This does not mean `num` could not be the result of an integer overflow. It simply reflects the partially intraprocedural dimension of our technique for detecting uses of overflown values.

```
void ex_false(
  uint num)
{
  if (num > 0x7fffffff) {
    uint siz = num * 2;
    S* tab = malloc(siz);
  }
}
```

```
void ex_true(
  uint num)
{
  uint siz =
    num + sizeof(H);
  if (siz < num) return;
  S* tab = malloc(siz);
}
```

```
void ex_dont_know(
  uint num)
{
  uint siz =
    num * sizeof(S);
  S* tab = malloc(siz);
}
```

Figure 12: Using an overflown value: different results

6.2 Categories of False Positives

Most integer overflows detected by our tool are false positives, and most of these false positives fall into one of the following categories, in decreasing order of importance:

1. casts between 32-bits signed value -1 and 32-bits unsigned value 0xffffffff, used to report an error status
2. casts of the status value returned by a function between 32-bits signed and unsigned integer types
3. casts of variables storing status values between 32-bits signed and unsigned integer types
4. explicit casts (introduced by the programmer, contrary to implicit casts introduced by the compiler)
5. loop decrements over an unsigned loop counter, like in `while(i--)`, which cause the loop to exit with an underflow on the loop counter (in dead code most of the time, meaning the counter is not used past the loop)

By default, we do not generate warnings for integer overflows that correspond to cases 1, 2, 4 and 5. These settings can be reversed individually on option. Because integer overflows corresponding to case 3 are more likely to uncover bugs, we still issue a warning for these, but we defined specific warnings for them, so that they can be easily recognized.

The remaining false positives fall in four categories: (1) they are justified integer overflows, as shown in Section 1; (2) they cannot arise in practice due to the some physical limitations (*e.g.*, the number of processors) that our approach does not take into account; (3) they depend on an invariant of the system (*e.g.*, the range of values of some field) that our approach is not aware of and (4) they stem from the imprecision of our tool.

6.3 Ranking of Warnings

Ranking is an essential component of PREfix in making bug-finding more effective. It computes a score for each warning, which is used to present warnings to the user in increasing order. Since the raw output of PREfix mixes highly probable defect reports with very unlikely ones, this improves the developer experience. Typically a cutoff score is used to separate the accurate warnings from the false positives.

To facilitate ranking and manual filtering, we generated different warnings for all checks presented in Section 5: 17 warnings correspond each to a different overflow/underflow check; 2 warnings correspond to the two dangerous uses of overflown values discussed in Section 6.1; 2 warnings correspond to casts between signed and unsigned integer types of the same size, in order to isolate the case 3 discussed in Section 6.2.

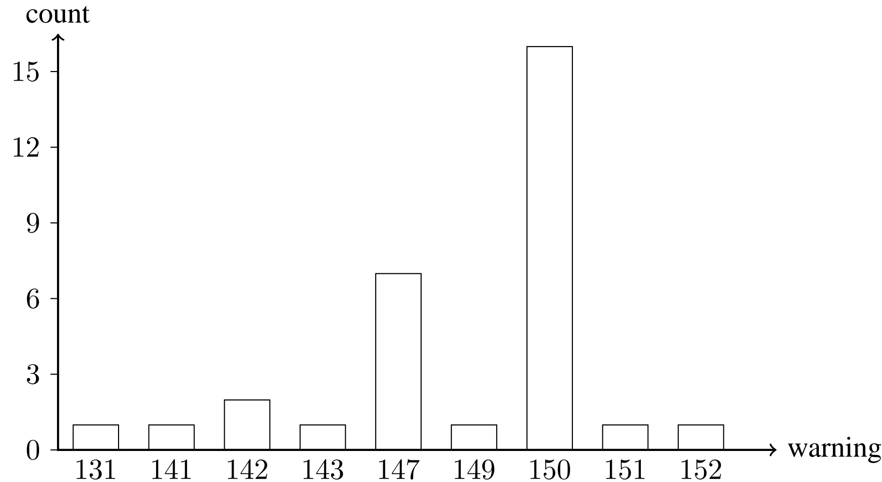


Figure 13: Count of bugs filed per warning (identified by a number)

Apart from the general heuristics applied by PREfix for every warning, depending on the locality, the kind of variables involved, *etc.*, specific rules penalize or promote warnings based on their category. We chose to penalize the following categories of overflow warnings, in decreasing order of importance:

1. Warnings that correspond to casts between signed and unsigned integer types of the same size are penalized most. It takes care of the case 3 discussed in Section 6.2.

2. Warnings that correspond to underflow in signed multiplication are penalized next, because (1) there are an awful lot of them and (2) they never uncovered a bug in our experiments.

3. Warnings that correspond to various underflows/overflows in arithmetic operations are slightly penalized, because (1) there are quite many of them and (2) they uncovered less bugs than other categories in our experiments. This concerns overflows on signed/unsigned addition and multiplication and underflow on signed subtraction. Other categories are not penalized because they generate fewer warnings, or, in the case of underflow on unsigned subtraction, we found that the corresponding warnings uncovered many real bugs.

7 Results on the Microsoft Product Code Base

We applied PREfix, with Z3 inside, to a substantial part of Microsoft's code base, consisting of over 10 million lines of C/C++ code. One of the authors spent three days reviewing integer overflow warnings, split among a few weeks. This allowed us to gain insight into the categories of expected integer overflows and false positive that we discuss in Section 1.1 and Section 6.2. Other warnings present fluctuations w.r.t. PREfix before the integration that are typical of even small modifications in such a complex tool when applied on such a large code base.

Figure 13 summarizes our results. We filed 31 bugs, 17 of which were generated by our security checks for uses of overflown values. Warning 150 which generates the most bugs (16) is the use of an overflown value in an allocation size. Warning 147 which generates 7 bugs is the underflow on an unsigned subtraction, which is why we did not penalize it along with other arithmetic checks in ranking.

Based on review from developers, the accuracy rate on these bugs is approximately 50%.

```
1  LONG l_sub(LONG l_var1, LONG l_var2)
2  {
3      LONG l_diff = l_var1 - l_var2; // perform subtraction
4      // check for overflow
5      if ( (l_var1>0) && (l_var2<0) && (l_diff<0) ) l_diff=0x7FFFFFFF;
6      ...
```
```
7  iElement = m_nSize;
8  if( iElement >= m_nMaxSize )
9  {
10     bool bSuccess = GrowBuffer( iElement+1 );
11     ...
12 }
13 ::new( m_pData+iElement ) E( element );
14 m_nSize++;
```
```
15 for (UNWORD uID = 0; uID < uDevCount && SUCCEEDED(hr); uID++) {
16     ...
17     if (SUCCEEDED(hr)) {
18         uID = uDevCount;  // Terminates the loop
```
```
19 ULONG AllocationSize;
20 while (CurrentBuffer != NULL) {
21     if (NumberOfBuffers > MAX_ULONG / sizeof(MYBUFFER)) {
22         return NULL;
23     }
24     NumberOfBuffers++;
25     CurrentBuffer = CurrentBuffer->NextBuffer;
26 }
27 AllocationSize = sizeof(MYBUFFER)*NumberOfBuffers;
28 UserBuffersHead = malloc(AllocationSize);
```
```
29 DWORD dwAlloc;
30 dwAlloc = MyList->nElements * sizeof(MY_INFO);
31 if(dwAlloc < MyList->nElements)
32     ... // return
33 MyList->pInfo = MIDL_user_allocate(dwAlloc);
```

Figure 14: Real integer overflow bug samples

Figure 14 shows a few sample code stubs containing integer overflow bugs that our tool discovered in the code base. These bugs illustrate how integer overflows manifest themselves in the real code, as opposed to select, well-known, samples from Figure 1 presented in Section 1.

Function `l_sub` performs safe subtraction of signed 32-bits integers by detecting overflows *a posteriori*. The problem is that the test on line 5 misses the case where `l_var1 == 0 && l_var == INT_MIN`. The first test `l_var1>0` should read in fact `l_var1>=0`. Thus, the programmer intended to check for overflows, wrote a custom routine, but the custom routine misses a case. Interestingly, this bug is identified using PREfix and Z3 because a call site is able to produce arguments that fall through the overflow check.

On lines 7-8, both fields `m_nSize` and `m_nMaxSize` could be equal to `UINT_MAX`, in which case the argument to `GrowBuffer` on line 10 is 0. Later on, the placement new on line 13 writes in unallocated memory.

On line 18, `uDevCount` could be `UINT_MAX`, in which case the attempt to terminate the loop by setting `uID` to `uDevCount` does not work, as `uID` is incremented to 0 at the end of the loop, before the test `uID < uDevCount`.

The test on line 21 does protect from an integer overflow in the multiplication

`sizeof(MYBUFFER)*NumberOfBuffers`, but as `NumberOfBuffers` is incremented on line 24 just before the loop exits, the test is ineffective.

On line 32, the test performed would be effective at detecting integer overflows on an addition, but it is unfortunately ineffective for detecting integer overflows on the multiplication `MyList->nElements * sizeof(MY_INFO)`.

8 Conclusion

To our knowledge, this is the first static analysis tool that seriously addresses the detection of integer overflow bugs in large legacy code bases. Yet the problem of integer overflows has been known for at least 30 years. Our tool addresses this challenge on programs written in C/C++, where most integer overflows are intended or benign. This increases the difficulty of our task in two ways: first, our tool must assume overflowing semantics for integers; secondly, our tool must distinguish intended uses of integer overflows from bugs.

We defined 17 non-overflowing checks as properties in the theory of bit-vectors. Each check ensures that the corresponding arithmetic or cast operation either does not overflow or does not underflow. We showed that the encoding of these propositions into bit-level circuits by bit-blasting generated a low number of clauses, except for checks on multiplication, for which we devised cheap approximate checks that retain most of the discriminating power of the real check. A result of our work was also to make these exact arithmetic overflow and underflow checks available as part of the Z3 API.

Apart from integer arithmetic specific checks, we defined checks for uses of overflown values which lead to buffer overflows, with harmful consequences for security. This allowed us to generate these warnings more aggressively, and to focus our manual review on these more critical issues.

From our experience at finding bugs in this code base, we classified the intended uses of integer overflows as intentional, reversed and checked, and we identified 5 broad categories of intentional integer overflows. This work lead to the definition of default settings and ranking strategies which concur to present the user of our tool with the most serious warnings first.

Although we started this work as a research experiment, the capabilities of PREfix with Z3 appear sufficiently mature to be exercised on the Microsoft code base. Ongoing work is to further reduce the number of false positives presented to the user, while retaining the most serious issues we detect with the current version. Apart from designing additional special categories of false positives, we expect important gains from a better use of the annotations already present in the code (originally for buffer overflows) and from the manual addition of models for standard functions manipulating strings, buffers or integers, both tailored towards integer overflow bug-finding.

There are many ways we could extend our tool to better detect integer overflow bugs. An obvious extension is to consider other dangerous uses of overflown values, *e.g.*, as the size of a call to `memcpy` [7]. Another idea is to exploit the many calls in the code to functions from the safe libraries SafeInt [39] and intsafe [29], that indicate the user intent to protect some sensitive values from otherwise possible and harmful integer overflows. It should be possible to design a tainting mechanism, such that all values that concur in the computation of a sensitive value, or that derive from such a value, are also marked as sensitive. Integer overflow warnings about these values would be considered highly critical.

Overall, we are confident that our goal of presenting the user with high-risk security bugs containing few false positives is within reach of a practical deployment. Thanks to the carefully designed incompleteness inherent to PREfix, and that we introduced in our integer overflow checks too, we managed to get a sound, efficient, bit-precise static analysis for detecting integer overflow bugs in the large.

Acknowledgments

We would like to thank Tom Ball, Leonardo de Moura, Mei Zhang, David Nanninga, Masood Siddiqi, Patrice Godefroid, Tim Fleehart, Mac Manson and Dennis Crain for their help and feedback.

References

[1] Domagoj Babić and Frank Hutter. Spear theorem prover. In *Proc. of the SAT 2008 Race*, 2008.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. *SIGPLAN Not.*, 37(1):1–3, 2002.

[3] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, LNCS 3362, pages 49–69. Springer, 2005.

[5] C. Barrett and C. Tinelli. CVC3. In *CAV '07*, 2007. to appear.

[6] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/download/acsl_1.4.pdf`.

[7] blexim. Basic integer overflows. *Phrack Magazine*, December 2002. `http://www.phrack.com/issues.html?issue=60&id=10`.

[8] Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. June 2006. `http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html`.

[9] Bryan A. Brady and Sanjit A. Seshia. The Beaver SMT solver. http://uclid.eecs.berkeley.edu/wiki/index.php/UCLID.

[10] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In Stefan Kowalewski and Anna Philippou, editors, *TACAS*, volume 5505 of *Lecture Notes in Computer Science*, pages 174–177. Springer, 2009.

[11] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. A lazy and layered SMT(BV) solver for hard industrial verification problems. In Damm and Hermanns [19], pages 547–560.

[12] Randal E. Bryant. A view from the engine room: Computational support for symbolic model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 145–149. Springer, 2008.

[13] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[14] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.

[15] CERT. Secure integer library, 2006. `http://www.cert.org/secure-coding/IntegerLib.zip`.

[16] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[17] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer Verlag, 2005.

[18] M. Costa, J. Crowcroft, M. Castro, A. I. T. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: end-to-end containment of internet worms. In *SOSP*, pages 133–147, 2005.

[19] Werner Damm and Holger Hermanns, editors. *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*. Springer, 2007.

[20] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS 08*, 2008.

[21] R. DeLine and K. R. M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report 2005-70, Microsoft Research, 2005.

[22] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[23] B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV'06*, LNCS 4144, pages 81–94. Springer-Verlag, 2006.

[24] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In Damm and Hermanns [19], pages 519–531.

[25] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[26] Mustafa Gök, Michael J. Schulte, and Mark G. Arnold. Integer multipliers with overflow detection. *IEEE Trans. Computers*, 55(8):1062–1066, 2006.

[27] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127, 2006.

[28] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 232–241, New York, NY, USA, 2006. ACM.

[29] Michael Howard. Safe integer arithmetic in C, February 2002. `http://blogs.msdn.com/michael_howard/archive/2006/02/02/523392.aspx`.

[30] ISO-IEC. *Programming Languages—C, ISO/IEC 9899:1990 International Standard*, 1990.

[31] ISO-IEC. *Programming Languages—C++, ISO/IEC 14882:1998 International Standard*, 1998.

[32] Franco Ivančić, Ilya Shlyakhter, Aarti Gupta, and Malay K. Ganai. Model checking c programs using F-SOFT. In *ICCD '05: Proceedings of the 2005 International Conference on Computer Design*, pages 297–308, Washington, DC, USA, 2005. IEEE Computer Society.

[33] Paul B. Jackson, Bill J. Ellis, and Kathleen Sharp. Using SMT solvers to verify high-integrity programs. In *AFM '07: Proceedings of the second workshop on Automated formal methods*, pages 60–68, New York, NY, USA, 2007. ACM.

[34] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall PTR, Englewood Cliffs, NJ, USA, first edition, 1978.

[35] Sumant Kowshik. How to defend against deadly integer overflow attacks. *eweek.com blog*, February 2009.

[36] S. K. Lahiri and S. Qadeer. Back to the Future: Revisiting Precise Program Verification using SMT Solvers. In *POPL'2008*, 2008.

[37] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jonathan D. Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[38] Gary T. Leavens, Clyde Ruby, K. Rustan, M. Leino, Erik Poll, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 105–106, New York, NY, USA, 2000. ACM.

[39] David LeBlanc. SafeInt, 2003. `http://www.codeplex.com/SafeInt`.

[40] David LeBlanc. Integer handling with the C++ SafeInt class. *MSDN Library*, January 2004. `http://msdn.microsoft.com/en-us/library/ms972705.aspx`.

[41] John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.

[42] D. Molnar P. Godefroid, M. Levin. Automated Whitebox Fuzz Testing. Technical Report 2007-58, Microsoft Research, 2007.

[43] PolySpace Client for C/C++ 7.0.1. 2009. `http://www.mathworks.com/`.

[44] S. Ranise and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2006.

[45] N. Tillmann and W. Schulte. Unit Tests Reloaded: Parameterized Unit Testing with Symbolic Execution. *IEEE software*, 23:38–47, 2006.

[46] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2:*

Classical Papers onComputational Logic 1967-1970, pages 466–483. Springer-Verlag, 1983.

[47] Robert Wille, Görschwin Fey, Daniel Große, Stephan Eggersglüß, and Rolf Drechsler. SWORD: A SAT like prover using word level information. In *VLSI-SoC*, pages 88–93. IEEE, 2007.