# Towards Efficient Dynamic Integer Overflow Detection on ARM Processors

Glenn Wurster
BlackBerry Limited
gwurster@blackberry.com

James Ward
BlackBerry Limited
jaward@blackberry.com

## ABSTRACT

In this paper, we present an approach to limiting the impact of integer overflow vulnerabilities by either trapping on integer overflow, or using saturating arithmetic. We focus our approach on efficient ARM processor detection of integer overflows at run-time using the current instruction set. We prototype efficient processor detection of integer overflow using modified GCC. Based on the initial prototype implementation and in-depth analysis, we identify current shortcomings in ARM processors when dealing with integer overflow. We propose minor tweaks in processor design to further increase performance.

## 1. INTRODUCTION AND OVERVIEW

Modern languages and compilers attempt to defend against specific classes of vulnerabilities through low overhead, minimally invasive approaches. Languages such as Java combat use-after-free vulnerabilities by including garbage collectors instead of requiring the developer to manage memory. Buffer overflows are handled in Python through enforcing strict limits on array indexes. Even programs written in C and C++, which don't have the same strict type safety checks, benefit from features such as stack cookies, address space layout randomization (ASLR), and non-executable data [14].

Integer overflows are one class of error that different languages attempt to combat in different ways. Java allows an integer to overflow, Python detects the overflow and converts the integer to a longer data type, and Swift throws an exception on overflow. In C/C++, some compilers (e.g. the GNU Compiler Collection (GCC) and Clang) support generating extra code to abort program execution when an overflow is detected [17]. Widespread integer overflow detection can introduce costly run-time overhead though, prohibiting its use in a performance critical production environment. Static analysis helps find possible integer overflows in languages such as C, but often involves specialized analysis engines run in development environments.

Regardless of the language, all strategies for dynamically mitigating the effects of integer overflow require detecting that an overflow has occurred (or will occur) and taking appropriate action. In this paper, we present two approaches to mitigating the effects of unintended integer overflows, with a focus on efficient run-time performance. Our primary approach relies on generating a signal whenever an overflow occurs. This signal allows the developer or language run-time to quickly pinpoint the source of an integer overflow and take corrective action. Depending on the language, handling the signal might result in conversion to a longer data type, or an exception being thrown. Our alternate approach relies on saturating arithmetic, preventing the overflow. While trapping has been attempted before [17, 7], we focus on improving the performance.

To demonstrate that our approaches are sound, we modify GCC to mitigate overflows on the ARM processor. We use the modified compiler to test a variety of applications on ARM. Our initial implementation and results provide the base for a proposal that would allow integer overflow detection to enjoy the same widespread deployment and minimal overhead as ASLR or other mitigations.

Section 2 gives background on integer overflow vulnerabilities and saturating arithmetic. Section 3 discusses our implementation of saturating and trapping integer overflows for ARM. Section 4 discusses the results of our prototype implementation. Section 5 discusses a potential method forward, leading to broad deployment of integer overflow mitigations. Section 6 discusses related work. We summarize in Section 7.

## 2. OVERVIEW AND BACKGROUND

We now discuss integer overflow vulnerabilities and saturating arithmetic.

### 2.1 Integer Overflow Vulnerabilities

Brumley et al. [7] classify integer overflows into three main types:

1. An integer wrapping vulnerability is caused when arithmetic is performed on an integer, with the result being too large to fit into the memory allocated. One example is attempting to add 1 to `UINT_MAX` (`0xFFFFFFFF`), which results in the value `0x100000000`. When truncated to 32-bits, the result is 0 (`0x00000000`), which is `0x100000000` modulo $2^{32}$ (i.e., bits 33 and above are discarded).

2. A signedness error exists when a signed integer is interpreted as unsigned, or vice versa. Since most platforms represent integers in two's compliment, the signed integer -2 (`0xFFFFFFFE`) on a 32 bit platform is the same as the unsigned value `UINT_MAX` - 1. Conversions between signed and unsigned integers is straightforward for any value in the range `[0, INT_MAX]`. Unless care is taken, casting values outside that range might lead to a vulnerability.
3. Truncation overflow exists when an integral type is cast into a narrower type (e.g., casting an integer into a short), causing a loss of information and an overflow if the number cannot be represented by the destination type. For example, if an `int` equal to 32768 were to be casted into a `short`, the result would be -32768.

While intentional integer overflows can be safely accounted for, unintentional cases have lead to severe security vulnerabilities for applications written in C. According to statistics obtained from the National Vulnerability Database (NVD) [30], more than half of integer overflow vulnerabilities reported have a score of 7-10 (high) on the CVSSv2 scale [25].

It is the use of an overflowed value in referencing or allocating memory that typically causes a vulnerability. In one example, the result of an overflow is used in the call to `malloc()`, allocating a buffer that is smaller than expected. Writing to this buffer beyond the allocated length can lead to an exploitable vulnerability. In a language such as Java, this scenario would not lead to a vulnerability – the overflow may occur, but attempting to access memory beyond the allocated array would result in an exception being raised.

Most developers continue to use custom pre- and post-condition checking in production code, even when using static and dynamic analysis tools to find the overflows. Custom pre- and post-condition checks are not always easy to reason about, and must be done without relying on undefined behavior [33, 34]. The C standard leaves signed integer overflow behavior undefined, and modern compilers may use that fact to optimize away certain security checks [16, 19].

Another issue surrounding integer overflow checking is that while the concept may seem relatively simple, writing proper checks for all vulnerable cases may be difficult. Zhang et al. [44] describe how an integer overflow vulnerability was patched in the CUPS printing system, but required a second patch to fully fix the issue.

## 2.2 Detection of Overflow in Languages

While Python 3 uses only unlimited precision integers [32], Python 2.7 only uses unlimited precision integers when overflow is detected. The `int_add` function is responsible for detecting whether addition overflowed with the `long` data type, storing the result to a longer data type if necessary. The `int_add` function uses XOR to detect overflow, which is reproduced in Listing 1. When compiled at -O2 in GCC, the overflow check used an additional four ARM instructions - two instructions to compute the XOR, and two branch instructions. The overflow check used three additional clock cycles on a MSM8974 processor.

In PHP 5.4.36, if an integer overflow occurs, the result is stored in a double [31]. Addition is implemented in the `add_function`, reproduced in Listing 2. When compiled at `-O2` in GCC, the overflow check used an additional four ARM instructions - two comparisons, and two branches, very similar to the ARM assembly produced for Python 2.7. The overflow check used three clock cycles on a MSM8974 processor.

```
static PyObject *
int_add(PyIntObject *v, PyIntObject *w)
{
  register long a, b, x;
  CONVERT_TO_LONG(v, a);
  CONVERT_TO_LONG(w, b);
  /* casts in the line below avoid undefined
     behaviour on overflow */
  x = (long)((unsigned long)a + b);
  if ((x^a) >= 0 || (x^b) >= 0)
    return PyInt_FromLong(x);

  return PyLong_Type.tp_as_number->
    nb_add((PyObject *)v, (PyObject *)w);
}
```

**Listing 1: Overflow detection in Python 2.7**

```
case TYPE_PAIR(IS_LONG, IS_LONG): {
  long lval = Z_LVAL_P(op1) + Z_LVAL_P(op2);

  /* check for overflow by comparing sign
     bits */
  if ((Z_LVAL_P(op1) & LONG_SIGN_MASK) ==
      (Z_LVAL_P(op2) & LONG_SIGN_MASK) &&
      (Z_LVAL_P(op1) & LONG_SIGN_MASK) !=
      (lval & LONG_SIGN_MASK))
  {
    ZVAL_DOUBLE(result,
                (double) Z_LVAL_P(op1) +
                (double) Z_LVAL_P(op2));
  } else {
    ZVAL_LONG(result, lval);
  }
  return SUCCESS;
}
```

**Listing 2: Overflow detection in PHP 5.4**

The Swift language [1] adds a trap and branch after each arithmetic assembly instruction. The standard arithmetic symbols (e.g., `+`) will cause an exception to be thrown if the operation would cause an overflow. In order to overflow without causing an exception, overflowing operators were introduced. The overflowing versions of the arithmetic operators include an ampersand prefix.

Both Java and C allow integer overflow to occur without throwing an exception. The C specification [23] states that signed integer overflow is undefined behavior, although most compilers will wrap. Some C/C++ compilers though, when configured, will detect overflow in integers [17, 7, 12] and cause a fault.

Regardless of the approach used to deal with integer overflow, many languages need to be able to quickly identify that an overflow has occurred and act [26]. Even if static analysis is used to reduce the number of checks, run-time checks cannot be entirely eliminated. In the remainder of this paper, we focus on fast dynamic mitigation of integer overflows.

## 2.3 A note on Saturation

Saturating arithmetic has been used in digital signal processing (DSP) and embedded programming, where integer wrapping can lead to significant signal distortion when processing analog signals. Saturating arithmetic prevents an overflow by forcing an arithmetic operation that would overflow to instead return the maximum (or minimum) value possible for the data type. As an example, multiplying 19 by 16 gives 304 which does not fit into a `char` (assuming a width of 8 bits). Without saturating, the result would overflow and return 48, but saturating multiplication would result in 127 (the largest value possible for a character).

Because saturating arithmetic forces variables to their minimum or maximum instead of overflowing, using a saturated value when referencing memory typically results in a memory violation (NULL pointer dereference or attempt to access privileged kernel memory). Integer overflows mitigated using saturating arithmetic are therefore turned from a potential vulnerability into (at worst), a denial of service.

Saturating arithmetic for integer overflow mitigation both borrows from failure oblivious computing [36], and also adds to it. Saturating arithmetic allows the application to continue running (in contrast to aborting execution immediately when an integer overflow is detected), while at the same time making the detection of subsequent memory errors more likely. Indeed, saturating arithmetic protects against CVE-2013-4511 [15], while at the same time allowing the kernel to continue running.

On ARMv6+ and later processors, the `QADD` and `QSUB` instructions support saturating addition and subtraction of signed 32-bit integers [2]. The `SSAT` and `USAT` instructions will saturate a value to a specific width, dealing with truncation overflow, but cannot account for overflow in the preceding arithmetic instruction. Because of hardware support and the potential to mitigate integer overflows, we include saturation alongside other methods that raise an exception on integer overflow.

## 3. IMPLEMENTATION

We first discuss our ARM assembly language integer overflow detection routines. We then discuss a prototype GCC implementation making use of the assembly routines. We present three different mitigations against integer overflows; branchless detection of overflows, branching when an overflow is detected, and using saturation. Including both branchless and branching forms of trapping allows us to determine whether the ARM processor can execute one faster than the other.

Our branchless trapping approach takes advantage of conditional execution, writing to memory pointed to by the program counter (`r15`) if the previous operation sets the overflow flag. On modern platforms, including Linux, program code is generally marked read-only, so the write will cause a SIGSEGV. If the application code has write enabled, our branchless code would need to be modified to avoid creating a vulnerability. During debugging, branchless trapping was useful in identifying the exact source line for an integer overflow in target programs.

Many of the routines update the flags register. When modifying GCC, we flagged the routines as clobbering the flags register, relying on GCC to perform the correct instruction reordering.

## 3.1 Signed Overflows

We first discuss handling of signed integer operations. For simplicity, registers are fixed in our examples. Our prototype implementation discussed below allows GCC to perform standard register allocation.

### *3.1.1 Addition & Subtraction*

Signed saturating addition and subtraction can be implemented with ARM's built-in saturating addition and subtraction instructions `QADD` and `QSUB` [2]. They saturate the result of $\texttt{r1} \pm \texttt{r2}$ into `r0` with the range $-2^{31} \leq \texttt{r0} \leq 2^{31} - 1$. Listing 3 contains our implementation of signed addition.

```
/* Conditional trap add */
adds   r0, r1, r2
strvs  r0, [r15]

/* Branching trap add */
adds   r0, r1, r2
bvs    .OVERFLOW

/* Saturating add */
qadd   r0, r1, r2
```

**Listing 3: ARM instructions for signed addition**

```
/* Conditional trap subtract */
subs   r0, r1, r2
strvs  r0, [r15]

/* Branching trap subtract */
subs   r0, r1, r2
bvs    .OVERFLOW

/* Saturating subtract */
qsub   r0, r1, r2
```

**Listing 4: ARM instructions for signed subtraction**

Subtraction is very similar for signed integers. In both cases, signed trapping is performed using either conditional execution or branching. Listing 4 contains our implementation for signed subtraction.

### *3.1.2 Multiplication*

Signed multiplication trapping can be done in three instructions [5], with saturation implemented in four instructions.

```
/* Conditional trap multiplication */
smull   r0, r3, r1, r2
cmp     r3, r0, asr #31
strne   r0, [r15]

/* Branching trap multiplication */
smull   r0, r3, r1, r2
cmp     r3, r0, asr #31
bne     .OVERFLOW

/* Saturating multiplication */
smull   r0, r3, r1, r2
cmp     r3, r0, asr #31
mvngt   r0, #0x80000000
movlt   r0, #0x80000000
```

**Listing 5: ARM instructions for signed multiplication**

The code in Listing 5 first multiplies `r1` and `r2` into a double wide integer using `SMULL`, storing the lower 32 bits in `r0` and the upper 32 bits in `r3`. It then compares the high bits to the low bits arithmetically right shifted by 31, using the `CMP` instruction. Shifting arithmetically will either set all bits if the high bit was set (i.e., it was a negative number), or will clear all bits if the high bit was clear (i.e., it was a positive number). If it is a negative result, the upper 32 bits of the multiplication result (which are in `r3`) must also be all set if the result did not overflow. If the result is positive, the upper 32 bits will all be cleared if there was not an overflow. For the saturating variant, we use the `MVN` and `MOV` instructions to set the result to either `0x7FFFFFFF` or `0x80000000` on overflow.

The implementation is loosely based off of GCC's `-ftrapv` multiplication test and a blog post [5].

### 3.1.3 *Left Shift*

In addition to supporting basic arithmetic, we have chosen to mitigate left shift. Because developers will sometimes use shift instead of multiplication, supporting both trapping and saturating on shifts might catch additional errors.

In the case of `r1 << r2`, the code in Listing 6 first counts the number of leading zeros in `r1` (using `CLZ`), comparing the result to `r2` (using `CMP`). If the number of leading zeros in `r1` is greater than the number stored in `r2`, a shift can be done without causing an overflow. Shifting a negative number or by a negative number is undefined in the C specification [23], and we choose to trap. In the saturating case, we saturate to `0x7FFFFFFF`. If programs depend on shifts using negative numbers, the assembly can be updated with additional instructions to support a different undefined behavior.

```
/* Conditional trap left shift */
clz      r3,  r1
cmp      r3,  r2
lslhi    r0,  r1,  r2
strls    r0,  [r15]

/* Branching trap left shift */
clz      r3,  r1
cmp      r3,  r2
lsl      r0,  r1,  r2
bls      .OVERFLOW

/* Left saturating shift */
clz      r3,  r1
cmp      r3,  r2
lslhi    r0,  r1,  r2
mvnls    r0,  #0x80000000
```

**Listing 6: ARM instructions for signed left shift**

```
/* Conditional truncate */
mov      r0,  r1, asr #7
teq      r0,  r1, asr #31
and      r0,  r1, #0xFF
strne    r0,  [r15]

/* Branching trap truncate */
mov      r0,  r1, asr #7
teq      r0,  r1, asr #31
and      r0,  r1, #0xFF
bne      .OVERFLOW

/* Saturating truncate */
ssat     r0,  #8,  r1
```

**Listing 7: ARM instructions for signed truncation to 8 bit**

We recognize that saturating a shift operator is controversial. The saturating shift operator results in bits being set in the result if an overflow occurs that would not have been set otherwise. If the developer is using the shift operator for bit manipulation, saturating will significantly alter the results. If, however, the developer was using the shift operator to perform multiplication, then saturating might be reasonable. We include saturating shift because it can overflow, but recognize that some developers will prefer to omit it.

### 3.1.4 *Truncation*

Truncating a variable from 32 bits to 8 bits is done by checking that bits 8 through 31 are all the same as the sign bit, illustrated in Listing 7.

### 3.1.5 *Casting to Unsigned*

Catching an overflow during a cast to unsigned involves checking whether the value being cast is less than 0 (or the high bit is set). The code we used is in Listing 8. A compare-negative instruction (`CMN`) is used in conjunction with a branch if carry set (`BCS`) so that Table 5 uses consistent branches.

```
/* Conditional cast */
cmp      r1, #0
mov      r0, r1
strlt    r0, [r15]

/* Branching trap cast */
cmn      r1,  #0x80000000
mov      r0,  r1
bcs      .OVERFLOW

/* Saturating cast */
cmp      r1,  #0
mov      r0,  r1
movlt    r0,  #0
```

**Listing 8: ARM instructions for casting signed to unsigned**

## 3.2 Unsigned Overflows

Catching overflows on unsigned arithmetic is similar to the approach for signed integers, except that the `QADD` and `QSUB` instructions cannot be used. Furthermore, the carry bit must be checked instead of overflow bit for most arithmetic operations.

### 3.2.1 *Addition & Subtraction*

As `QADD` does not support unsigned integers, the code in Listing 9 needs to explicitly saturate to `UINT_MAX` on overflow. The same applies for unsigned subtraction, except that we saturate to 0 in Listing 10:

```
/* Conditional trap addition */
adds     r0,  r1,  r2
strcs    r0,  [r15]

/* Branching trap addition */
adds     r0,  r1,  r2
bcs      .OVERFLOW

/* Saturating addition */
adds     r0,  r1,  r2
mvncs    r0,  #0
```

**Listing 9: ARM instructions for unsigned addition**

```
/* Conditional trap subtraction */
subs    r0, r1, r2
strcc   r0, [r15]

/* Branching trap subtraction */
subs    r0, r1, r2
bcc     .OVERFLOW

/* Saturating subtraction */
subs   r0, r1, r2
movcc  r0, #0
```

**Listing 10: ARM instructions for unsigned subtraction**

For both unsigned addition and subtraction, we can use the carry flag to determine whether we have overflowed during a calculation and update the result accordingly.

### 3.2.2 *Multiplication*

Unsigned multiplication is performed similar to the signed variant.

```
/* Conditional trap multiplication */
umull   r0, r3, r1, r2
cmp     r3, #0
strne   r0, [r15]

/* Branching trap multiplication */
umull   r0, r3, r1, r2
cmp     r3, #0
bne     .OVERFLOW

/* Saturating multiplication */
umull   r0, r3, r1, r2
cmp     r3, #0
mvnne   r0, #0
```

**Listing 11: ARM instructions for unsigned multiplication**

The code in Listing 11 first multiplies `r1` and `r2`, storing the resulting high bits into `r3` and low bits into `r0` using the `UMULL` instruction. Unlike the signed variant, overflow occurred if any of the bits in `r3` are non-zero, and is checked using the `CMP` instruction.

### 3.2.3 *Left Shift*

```
/* Conditional trap left shift */
clz     r3, r1
cmp     r3, r2
lslcs   r0, r1, r2
strcc   r0, [r15]

/* Branching trap left shift */
clz     r3, r1
cmp     r3, r2
lsl     r0, r1, r2
bcc     .OVERFLOW

/* Saturating left shift */
clz     r3, r1
cmp     r3, r2
lslcs   r0, r1, r2
mvncc   r0, #0
```

**Listing 12: ARM instructions for unsigned left shift**

Unsigned left shift is implemented the same as signed left shifting, with the exception that we saturate to `UINT_MAX` instead of `INT_MAX` in Listing 12. We count the leading zeros in the value we are shifting (using `CLZ`), and compare that to the amount we need to shift by. If the number of leading zeros is greater than or equal to the amount we are shifting by, the operation will not overflow. The trapping version replaces the conditional move with a `BCC` instruction.

### 3.2.4 *Truncation*

Truncating a variable from 32 bits to 8 bits is done by checking that the input value is less than 256. Listing 13 use code to compare the input to 256, trapping or branching after performing the actual truncation using an `AND` operation.

```
/* Conditional truncate */
cmp     r1, #0xFF
and     r0, r1, #0xFF
strhi   r0, [r15]

/* Branching trap truncate */
cmp     r1, #0x100
and     r0, r1, #0xFF
bcs     .OVERFLOW

/* Saturating truncate */
usat    r0, #8, r1
```

**Listing 13: ARM instructions for unsigned truncation to 8 bit**

### 3.2.5 *Casting to Signed*

Catching an overflow during a cast to signed involves checking whether the high bit is set (which would result in a negative signed integer). The code is illustrated in Listing 14.

## 3.3 Adding Integer Overflow Checking to GCC

While dynamic integer overflow checks are required in many different programming languages, we choose to concentrate on C for our prototype implementation. Many interpreters for other languages are written in C [26], and the majority of integer overflow vulnerabilities have affected applications written in C.

```
/* Conditional cast */
tst     r1, #0x80000000
mov     r0, r1
strne   r0, [r15]

/* Branching trap cast */
cmn     r1, #0x80000000
mov     r0, r1
bcs     .OVERFLOW

/* Saturating cast */
tst     r1, #0x80000000
mov     r0, r1
mvnne   r0, #0x80000000
```

**Listing 14: ARM instructions for casting unsigned to signed**

To evaluate our detection operations, we modified GCC to use the techniques discussed above. For completeness, we implemented both the trapping and saturation variants on both signed and unsigned integer arithmetic operations, including left shift. Test applications from the performance section were compiled with overflow mitigation by using a newly-defined command line flag.

Our decision to allow saturating and trapping of unsigned integers will no doubt be a controversial one, even if our end-goal is to evaluate run-time integer overflow performance for all languages. The C standard specifies that "computation involving unsigned operands can never overflow, because a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type [23]." We chose to implement mitigation for unsigned integers for the following reasons:

- CVE-2013-4511 [15] involved code using unsigned integers. A solution only affecting signed variables would require changing variable types in an already-written application in order to enable protection

- Because interpreters for other languages are often written in C, and might use different unsigned integer overflow semantics.

Overflow mitigation attributes are applied to each variable individually and tracked throughout the compilation process, extending on previous work in GCC that supports saturating fixed point arithmetic. Using a combination of new command line flags, a new `pragma`, and a new variable attribute, fine-grained mitigations can be applied at the per-variable level if required. The mitigation logic is implemented in the back end by extending the ARM register transfer language (RTL) with new instruction patterns called `insn`s. Mitigating and wrapping variables may be used together. During testing, mitigations were typically enabled using the command line flag as opposed to using the `pragma` or variable attributes.

The choice between trapping using a branch, trapping using conditional execution, and saturating is selected in our prototype by the command line flag `-fioverflow-type=<method>`. The exception raising code for our branch implementation generated a SIGSEGV by accessing the memory at address 0, and is shown in Listing 15.

Signed regular and modulo division can only overflow if performing the operation `INT_MIN / -1` and `INT_MIN % -1` respectively. Signed negation can only overflow when negating `INT_MIN`. We are not aware of any vulnerabilities caused by overflow on division, omitting these operations from the GCC prototype.

```
.OVERFLOW:
    /* Generate SIGSEGV */
    mov r0, #0
    str r0, [r0, #0]
```

**Listing 15: Branch target for overflow trap**

## 4. RESULTS

This section focuses on the results of our implementation. We discuss whether the mitigation approaches would prevent against known vulnerabilities, performance of our implementation, areas of intentional overflows, and the limitations of our current implementation.

### 4.1 Performance

To test performance, we added a `-fioverflow-type=count` argument to GCC. When enabled, each arithmetic instruction output by the compiler also output a new assembly instruction `ioc #<index>`. Binutils was updated to recognize the `ioc` ARM assembly instruction and output appropriate machine code. QEMU was configured to recognize the instruction and increment a counter indexed by the operand to `ioc`. Using this approach, we were able to count how many of each arithmetic operation was performed in the actual running program. Table 1 lists the percentage of instructions executed for each operator on each application.

We also created a synthetic benchmarking test application which measured the overhead of each type of arithmetic operation under each mitigation on an MSM8974 processor, listing our results in Table 2. Using the data from Table 1 and 2, we estimate the performance overehad for each application in 1. Even though `-ftrapv` does not affect unsigned instructions, we included it for comparison. We are significantly faster than `-ftrapv` in all but the `openssl` case. The performance numbers for `openssl` are worse because most arithmetic operations were unsigned, which `-ftrapv` does not mitigate.

We also ran the dhrystone benchmark on the MSM8974 processor under all different mitigation methods, listing our results in Table 3. Our performance results with dhrystone are better than those calculated in Figure 1 due to factors such as processor pipelining and branch prediction. We also include results using Clang 3.4's `-ftrapv` and `-fsanitize` for signed and unsigned integer overflows.

| Mitigation | Result |
|---|---:|
| Base Case | 10,060,362 |
| Saturating | 9,199,632 |
| Trapping | 9,233,610 |
| Branching | 9,960,159 |
| gcc `-ftrapv` | 5,136,107 |
| clang `-ftrapv` | 4,118,616 |
| clang `-fsanitize` | 4,258,944 |

**Table 3: Dhrystone Performance**

While dhrystone benchmark overheads of between 1% and 8.5% are significantly better than the 50% overhead of `-ftrapv`, in some environments those performance numbers will still be unreasonable. We therefore conclude that processor support is necessary to achieve significant further reductions in overhead, and discuss potential improvements to performance overhead in Section 5.

### 4.2 Intentional Overflows

On occasion, developers will intentionally cause integer overflows. A linear equation based pseudo-random number generator will often truncate the result to $2^{32}$ by allowing integer overflow in the arithmetic operations. Overflows are also used intentionally in floating point emulation [16] and when computing large modulus [7].

Our primary approach of branching to exception raising code whenever an overflow occurs was fastest. We also found during debugging that the branchless detection made intentional overflows easier to identify. Both approaches prevented subtle undetected errors from being introduced. In addition to testing saturation with our synthetic benchmark

| Title | signed + | signed − | signed ∗ | signed << | unsigned + | unsigned − | unsigned ∗ | unsigned << |
|---|---|---|---|---|---|---|---|---|
| dhrystone | 2.65% | 0.79% | 0.26% | 0.00% | 0.53% | 0.00% | 0.00% | 0.00% |
| gzip | 0.20% | 0.51% | 0.00% | 0.20% | 1.35% | 4.28% | 0.05% | 0.74% |
| gunzip | 0.44% | 0.71% | 0.00% | 0.00% | 2.96% | 2.17% | 0.89% | 0.00% |
| tar | 6.21% | 0.00% | 0.00% | 0.00% | 1.51% | 0.78% | 0.15% | 0.13% |
| untar | 10.48% | 0.17% | 0.01% | 0.00% | 0.16% | 0.03% | 0.00% | 0.00% |
| ffmpeg | 9.84% | 0.83% | 3.83% | 0.31% | 0.01% | 0.00% | 0.01% | 0.01% |
| openssl | 0.23% | 0.85% | 0.04% | 0.00% | 4.65% | 0.54% | 1.74% | 3.10% |
| sqlite | 1.20% | 0.65% | 0.29% | 0.00% | 0.76% | 0.16% | 0.10% | 0.16% |

**Table 1: Percentage of instructions executed**

| Title | Base | Saturating | Trapping | Branching | Ftrapv |
|---|---|---|---|---|---|
| signed + | 100.00% | 100.00% | 103.00% | 103.00% | 128.44% |
| signed - | 100.00% | 100.00% | 102.99% | 101.50% | 128.45% |
| signed * | 100.00% | 121.90% | 117.97% | 113.47% | 159.86% |
| signed << | 100.00% | 107.00% | 104.66% | 104.65% | 100.59% |
| unsigned + | 100.00% | 102.66% | 102.22% | 101.33% | 100.01% |
| unsigned - | 100.00% | 102.65% | 101.32% | 101.32% | 100.00% |
| unsigned * | 100.00% | 111.95% | 115.93% | 107.97% | 103.98% |
| unsigned << | 100.00% | 111.64% | 104.66% | 102.49% | 100.44% |

**Table 2: Run-time overhead for each operator and mitigation technique, with the operator being 1% of total executed instructions**

and dhrystone, we ran `tar` and `gzip` with saturation enabled and did not encounter any errors. Raising an exception, however, is more useful in pinpointing intentional overflows.

## 4.3 Limitations of the Prototype

Our current implementation only applies to 32-bit types. Applications that perform a majority of their calculations with other data types (e.g., 8, 16 or 64bit values) will not have mitigations applied to them. Our implementation can be extended to types with other lengths through additional RTL instructions. Expanding support to include types other than 32bit would also allow our implementation to support mismatched types, including truncation overflows, as described in Section 2.1.

During testing, a number of overflows were found in the binaries. The `ffmpeg` tool shifts a negative number, which is undefined behavior according to the C specification. Several of the tools also overflowed counters. While none of the errors found were security vulnerabilities, fixing pre-existing overflows was an iterative process. Finding and fixing overflows could be improved in the future by keeping track of the location of each exception instead of immediately aborting on integer overflow.

Our prototype always includes mitigations for all arithmetic operations, including shift. Disabling mitigations on operators like left shift would provide the developer with greater flexibility.

## 5. BUILDING ON THE SOLUTION

The minimum number of additional instructions for each operation are given in Table 4, with additional clock cycles for the MSM8974 in brackets. Tests were performed in a tight assembly loop using the performance monitor cycle count register [4]. We include integer truncation and signedness operations in our synthetic tests, even though our GCC prototype did not mitigate these operations. We chose casting from 32 to 8 bit as a representative sample. Several clock cycles were added for multiplication and shifting.

|  | Signed |  |  | Unsigned |  |  |
|---|---|---|---|---|---|---|
| OP | Trap Cond. | Trap Branch | Sat | Trap Cond. | Trap Branch | Sat |
| + | 1 (1) | 1 (1) | 0 (0) | 1 (1) | 1 (1) | 1 (1) |
| − | 1 (1) | 1 (1) | 0 (0) | 1 (1) | 1 (1) | 1 (1) |
| ∗ | 2 (4) | 2 (3) | 3 (3) | 2 (3) | 2 (2) | 2 (1) |
| << | 3 (1) | 3 (2) | 3 (3) | 3 (1) | 3 (1) | 3 (3) |
| Cast | 3 (2) | 3 (2) | 0 (0) | 2 (1) | 2 (1) | 0 (0) |
| Sign | 2 (1) | 2 (1) | 1 (1) | 2 (1) | 2 (1) | 2 (1) |

**Table 4: Additional Assembly Instructions (Additional Clock Cycles)**

Even though our approach is specific to the ARM architecture, our attempt to perform fast overflow checking on integer arithmetic has lead us to several additional observations about processor architecture:

- While a processor will generate a software interrupt on divide by zero and invalid memory accesses, the option to generate a software interrupt on overflow is limited to MIPS and ALPHA architectures [27, 13]. On all other architectures, the application must combine each arithmetic operation with subsequent jump and interrupt instructions to trap an overflow.

- While the standard addition and subtraction instructions on ARM allow immediate operands, the saturating versions do not. In practice, the compiler reordered and combined constants in order to minimize the performance impact. During our performance testing, we saw a 3% increase in the number of instructions executed by removing immediate operands.

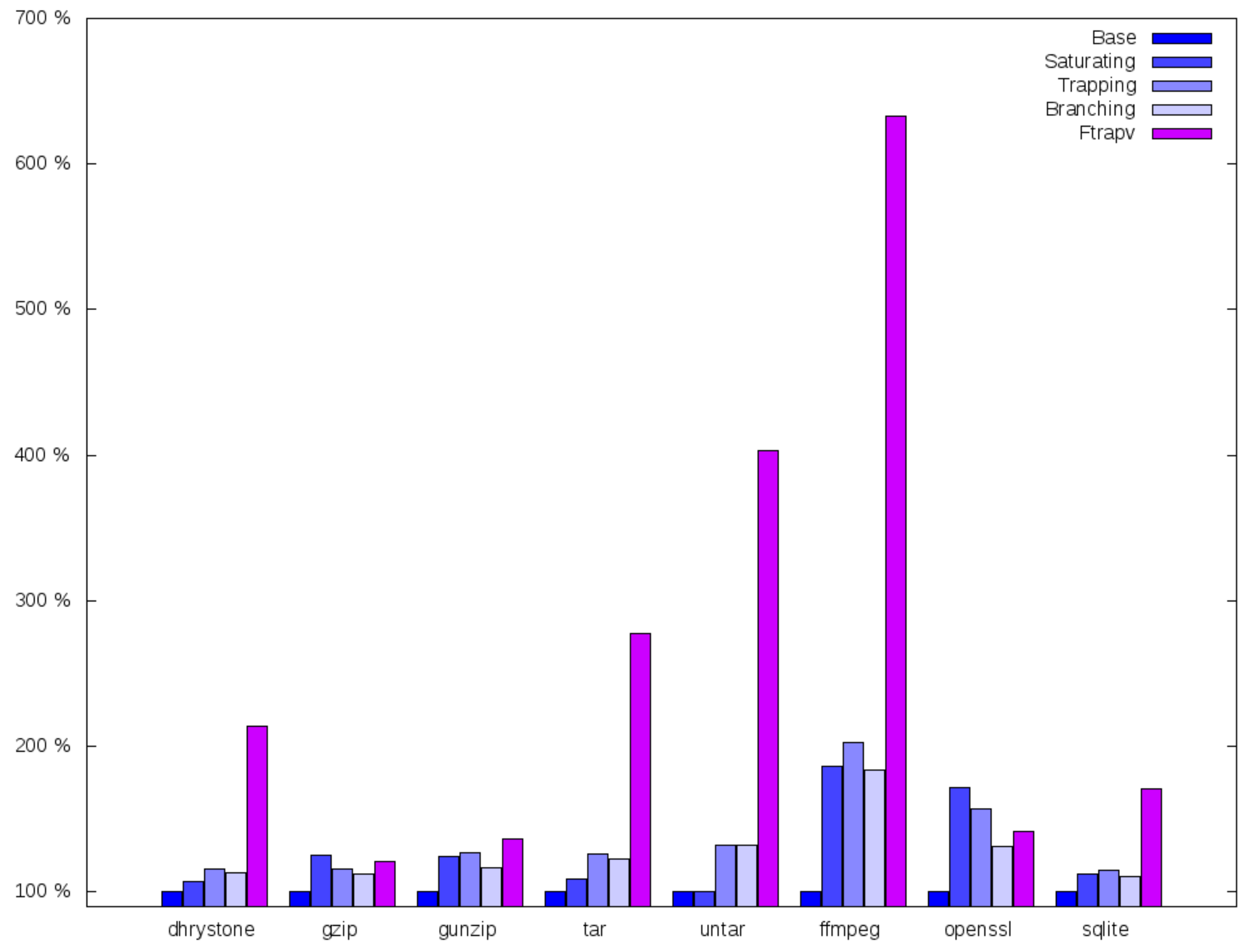

**Figure 1: Expected Performance Overhead**

- On ARM, updating the condition codes during arithmetic is optional. To avoid pipeline stalls, the GCC compiler will often re-order instructions, placing an arithmetic instruction which does not update the flags register between a compare (which does update the flags register) and subsequent branch. Because the flags must be set in order to detect overflow, our mitigations reduce the flexibility of the compiler in re-ordering instructions. Furthermore, the ARMv4 and earlier processors corrupt the carry and overflow flags on multiplication [2], so these instructions were avoided.

- The ARM instruction set has a QC (saturating) flag [2]. The QC flag is sticky, only being cleared by the `MSR` instruction. This flag would allow consolidating overflow checks for signed arithmetic after a sequence of arithmetic operations, and might improve performance. We did not implement this approach in our prototype because of extensive compiler changes required. The `QADD` and `QSUB` instructions set the QC flag in addition to saturating the result. The `SMLA` and `SMLAW` instructions update the QC flag, but do not saturate the result, and do not work with two 32bit operands.

- A processor does not always have sufficient information to determine whether an integer overflow has occurred. The same instruction is often used for both signed and unsigned arithmetic. Instead, the processor sets both the overflow and carry bits, which must be checked by the application after each arithmetic operation that might overflow. The carry bit indicates overflow in unsigned operations, and the overflow bit indicates signed integer overflow. The flags must be checked after each arithmetic operation because they will be reset with the next instruction that updates the flags. This adds significant overhead when implementing trapping arithmetic.

## 5.1 A Proposed Path Forward

Processors can easily keep track of when integer overflow happens, and will indicate an overflow by updating the status register flags. Unfortunately, the method chosen to convey this information to the application is inefficient, requiring a check after each arithmetic operation. In order to improve performance, others have chosen to use static analysis and similar approaches to limit the number of overflow checks. Many languages, however, including Python, Swift, and PHP, continue to require fast dynamic detection of integer overflows.

Modifying processor architecture to generate a software exception would improve performance overhead significantly, and would benefit all languages that currently attempt to detect and mitigate integer overflows. Unfortunately, the lack of awareness by the processor about the type of arithmetic being performed (signed or unsigned) appears to be the largest roadblock. On x86 [21], instruction prefixes could be created to indicate that the following arithmetic instruction should trap on signed or unsigned overflow. There does not, however, appear to be space in the current fixed-width instruction set of ARM to distinguish between signed and unsigned operators with the current arithmetic instructions. As an alternative, the processor could have a signed/unsigned mode flag, which could be updated by the application as required (i.e., set once, before a set of signed integer operations).

An alternative modification, which leaves the instruction set relatively untouched, is a slight variation of our branching mitigation. Rather than attempting to add many new instructions to the architecture, we tweak a few instructions to update the flags properly, allowing us minimize block length and take advantage of current and future branch prediction improvements. For ARM, this involves the following steps:

- Either create a new multiplication instruction, or modify the current multiplication instruction to properly update overflow and carry flags when requested. In Table 5, we assume the `MULS` instruction is updated.

- Create a left shift instruction (e.g., arithmetic shift left, or `ASL`) which updates the overflow flag if any set bits are shifted out of the register.

- Update the division and modulo division instructions to account for overflow and set the flags register if requested. Recall from above that signed regular and modulo division can only overflow if performing the operation `INT_MIN / -1` and `INT_MIN % -1` respectively.

- In hardware or microcode, optimize the branch prediction and instruction merging for a forward branch checking overflow or carry following an arithmetic instruction that sets the flags register. Assume the branch is not taken.

Our proposed assembly language instructions for the ARM processor, taking into account a modified multiplication and shift instruction, are shown in Table 5. For consistency, we propose always branching to a label at a positive offset on overflow, and not branching if an overflow does not happen. The addition and subtraction operations are backwards compatible with current processors, but the multiplication and shift operators rely on new instructions.

In higher level languages such as C, having the ability for the compiler to detect and respond to overflows might improve program performance in the long-run. Instead of performing arithmetic twice (the first time to detect if an overflow will occur while avoiding undefined compiler behavior [33, 34, 40]), the check can be replaced with an appropriate signal handler or other language construct, and the arithmetic performed once. The C language does not have a single uniform way to perform an integer overflow check, although several safe integer libraries have been developed [22, 37, 18]. For the vast majority of code that does not use a safe integer library, all developers must currently write the overflow check manually.

| Operation | ARM Template |
|---|---|
| Signed $+$ | `adds .*`<br>`bvs +.LABEL` |
| Signed $-$ | `subs .*`<br>`bvs +.LABEL` |
| Signed $*$ | `muls .*`<br>`bvs +.LABEL` |
| Signed $<<$ | `asl .*`<br>`bvs +.LABEL` |
| Signed $\div$ | `sdiv .*`<br>`bvs +.LABEL` |
| Signed % | Not supported on ARM |
| Signed Cast (8bit) | `mov rTMP, r?, asr #7`<br>`teq, rTMP, r?, asr #31`<br>`bne +.LABEL` |
| Signed $\rightarrow$ Unsigned | `cmn r?, #0x80000000`<br>`bcs +.LABEL` |
| Unsigned $+$ | `adds .*`<br>`bcs +.LABEL` |
| Unsigned $-$ | `subs .*`<br>`bcc +.LABEL` |
| Unsigned $*$ | `muls .*`<br>`bcs +.LABEL` |
| Unsigned $<<$ | `asl .*`<br>`bcs +.LABEL` |
| Unsigned $\div$ | Cannot overflow |
| Unsigned % | Cannot overflow |
| Unsigned Cast (8bit) | `cmp r?, #0x100`<br>`bcs +.LABEL` |
| Unsigned $\rightarrow$ Signed | `cmn r?, #0x80000000`<br>`bcs +.LABEL` |

**Table 5: Proposed standardized integer overflow checks for ARM**

## 5.2 Further Work

Our implementation has also shown that overflow detection can be implemented efficiently, but requires hardware changes for future performance improvements. Our research has also shown that applying trapping to integer operations can mitigate or reduce the severity of an integer overflow vulnerability. Our implementation and any processor improvements resulting from our research can benefit numerous higher languages such as PHP, Python, C, C++, and Swift. Further research of processor improvements is future work.

Another way to implement saturation on ARM is to utilize the NEON co-processor if it is available. NEON is a separate instruction set designed for use in media and batch processing. NEON has built-in saturation instructions, for both signed and unsigned at various data widths. This would seem to be an ideal way to have implemented saturation, however there is additional overhead involved in transferring data to NEON [3]. We have not measured the cost of using the NEON core, instead focusing on core ARM assembly in this paper.

While this paper focused on the ARM instruction set, expanding our GCC implementation to work on x86 or other processors would involve simply implementing the appropriate instruction patterns (`insns`) for x86. Expanding our prototype to work with widths other than 32bit is future work.

## 6. RELATED WORK

Safe integer libraries like IntSafe [22] and SafeInt [37] aim to reduce the number of integer overflow vulnerabilities by providing safe arithmetic functions or macros. Similarly, GCC 5 includes three built-in functions for performing addition, subtraction, and multiplication [18]. The built-in functions return `true` if an overflow occurred. Our suggested standardized processor instructions could be used to increase the performance of all libraries. Safe integer libraries allow for fine grain control. Unfortunately, they also require that a library be chosen early in development or that significant re-architecting of legacy code be performed.

There has also been tremendous work in improving the performance and accuracy of static analysis tools [42, 8, 9, 28, 29, 6]. Wang et al. [42] attempt to statically detect integer overflows. Their approach uses multiple analysis phases to catch invalid invariants, negative array indices, and traditional integer overflows. Their approach reduces false positives by using taint analysis in order to identify possible vulnerable areas of the code. Programmers are still required to create rules for the analyzer to white-list certain types of overflows that should not be caught in order to reduce false positives. Microsoft has employed a static analysis tool [29] that mixes symbolic execution with constraint solving to attempt to analyze code at scale to find integer overflow vulnerabilities. Their tool also encounters issues with branch analysis and loop unrolling, leading to false negatives. These tools are designed to be used during development. One benefit of efficient dynamic detection of integer overflows in production environments is decreased effort and time spent during development. Furthermore, checks that cannot be statically analyzed can be implemented efficiently at run-time using our approach.

Tools such as KLEE [8], EXE [9], and SmartFuzz [28] use symbolic execution techniques to inspect code, generating fuzzing test cases that will trigger vulnerable conditions. Programs analyzed using these tools will benefit from our work for run-time overflow checks that need to be added.

Binary analysis tools [41, 43] have also been used to find integer overflows. In IntScope [41], binaries are disassembled and the program flow is simulated. IntScope attempts to match up sources of tainted data (e.g. `fread` and `recv`) with vulnerable sinks (e.g., `malloc` and `alloca`), checking the execution path for overflows. The analysis engine has to determine the difference between intentional and unintentional overflows added by developers, and overflows that might be generated by the compiler. Assembly code also loses type information (e.g., signed vs̀unsigned), forcing the tool to infer types. We do not attempt to infer type information.

There have been a number of run-time analysis tools [20, 7, 17, 16, 11, 10] for detecting integer overflow. Most only attempt to detect signed integer overflow. GCC's `-ftrapv` [17] add checks during compilation in order to catch signed integer overflows during run-time. Others, like BRICK [10] hook into the program flow through other means. In this paper, we address remaining performance issues in dynamic detection and expand coverage. Pre-existing tools could be easily modified to use our standardized assembly instructions, receiving the same performance boost.

GCC, when used with the `-ftrapv` flag, replaces signed integer operations with function calls into standard library functions. Unfortunately, `-ftrapv` does not implement checks for unsigned integers or shifts, and has no way of detecting the difference between intentional and unintentional overflows. During our performance testing, `ftrapv` was found to be significantly slower than our approach.

Clang's `-fsanitize-undefined-behavior` compiler flag includes the Integer Overflow Checker(IOC) [16], and provides protections similar to `-ftrapv`. IOC was not designed for use in production systems, as it incurs a mean performance overhead of over 50%. BRICK [10] is another tool designed for run-time checking during testing. It is built on top of Valgrind [39], and similarly adds a performance overhead of about 50%.

BLIP [20] is a run-time tool that adds overflow checks to loop variants in an attempt to reduce buffer overflows caused by large loops. The patch is limited to only detecting overflows in loop structures and is not suitable for generically detecting integer overflows.

RICH [7, 6] defines formal semantics for integers by applying sub-typing theory and defining rules for safe integer operations. Integer casting is covered in their implementation, and integer overflows are caught by up-casting the operation and later down-casting the result – following the sub-typing rules defined earlier. Zhang et al. [7] report an average performance overhead of 3.7%. Our implementation might be able to increase the performance even more.

IntPatch [44] combines taint-analysis and a custom type system to reduce the number of checks that need to be inserted into a program, reducing the performance overhead. IntPatch relies on program slicing to reduce the number of false positives, and reports better performance results than RICH, at less than 1% during run-time. The analysis phase of IntPatch takes a significant amount of time.

A change to the integer model in C is proposed by CERT to adopt a model titled "as-if infinitely ranged integers" [24]. Their proposed model has the compiler automatically handle integer overflows by representing values as if they were calculated using an infinite integer model, or to trap upon execution. Our work in this paper would help the performance overhead of their proposal.

Regehr argues that hardware traps for integer overflow are needed [35]. In this paper, we propose one method for hardware and software to work together, improving the performance of integer overflow trapping to mirror a hardware implementation. Implementing traps in hardware would decrease the size of an application compared to arithmetic followed by branches, but would require significant changes in order for the hardware to understand the difference between signed and unsigned arithmetic. Bramley [5] argued that the performance overhead of the branching trap forms of multiplication and addition discussed above should be minimal, however we have shown there is still progress to be made.

## 7. CONCLUSION

We have introduced assembly language constructs for mitigating integer overflow which support further optimization by hardware vendors. Our processor optimizations are designed to be fast and broadly applicable across many languages and previous integer overflow protection implementations. Our approach mirrors what stack cookies [14], ASLR [38], and non-executable data memory attempt – a class of programming errors previously exploitable is changed into a trap and program abort, with minimal performance overhead.

History has shown that tools which can be enabled by a simple compiler option to provide benefit to pre-existing code bases are much more likely to be broadly deployed than solutions that require a change in developer behavior. With this in mind, we concentrated on fast and simple run-time checks that do not depend on static analysis, and can run on production code. While processor improvements will continue to reduce the performance overhead of our approach, this paper presents a significant step toward a broadly deployable solution, regardless of the language.

## 8. REFERENCES

[1] *The Swift Programming Language*, 2014th ed., Apple Inc., Jun 2014, https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/.

[2] *ARM Architecture Reference Manual, ARM v7-A and ARMv7-R edition*, ARM Limited, Apr 2008, aRM DDI 0406B.

[3] *Cortex-A15 Technical Reference Manual*, ARM Limited, May 2010.

[4] *Cortex-A8 Technical Reference Manual*, ARM Limited, May 2010.

[5] J. Bramley, "Detection overflow from MUL," Web Page (accessed 29 Jul 2014), Aug 2010, http://community.arm.com/groups/processors/blog/2010/08/25/detecting-overflow-from-mul.

[6] D. Brumley, D. Song, and J. Slember, "Towards automatically eliminating integer-based vulnerabilities," Carnegie Mellon University, Tech. Rep. CMU-CS-06-136, Mar 2006, http://reports-archive.adm.cs.cmu.edu/anon/2006/CMU-CS-06-136.pdf.

[7] D. Brumley, C. Tzi-cker, R. Johnson, H. Lin, and D. Song, "RICH: Automatically protecting against integer based vulnerabilities," in *Proceedings of the 14th Annual Network & Distributed System Security Symposium*, Feb 2007.

[8] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, Dec 2008, pp. 209–224.

[9] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, Oct 2006, pp. 322–335.

[10] P. Chen, Y. Wang, Z. Xin, B. Mao, and L. Xie, "BRICK: A binary tool for run-time detecting and locating integer-based vulnerability," in *Proceedings of the 4th International Conference on Availability, Reliability and Security*, Mar 2009, pp. 208–215.

[11] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya, "ARCHERR: Runtime environment driven program safety," in *Proceedings of the 9th European Symposium on Research in Computer Security*, Sep 2004, pp. 385–406.

[12] "Clang compiler user's manual," Web Page (accessed 10 Aug 2015), Aug 2015, http://clang.llvm.org/docs/UsersManual.html.

[13] *Alpha Architecture Handbook*, Version 4 ed., Compaq Computer Corporation, Oct 1998, http://www.compaq.com/cpq-alphaserver/technology/literature/alphaahb.pdf.

[14] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Expo*, Jan 2000, pp. 119–129.

[15] "Multiple integer overflows in alchemy lcd frame-buffer drivers in the linux kernel," Web Page (accessed 09 Feb 2015), http://www.cvedetails.com/cve/CVE-2013-4511/.

[16] W. Dietz, P. Li, J. Regehr, and V. Adve, "Understanding integer overflow in C/C++," in *Proceedings of the 34th International Conference on Software Engineering*, Jun 2012, pp. 760–770.

[17] "Options for code generation conventions," Web Page (accessed 28 Jul 2014), https://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html.

[18] "GCC 5 release serise changes, new features, and fixes," Web Page (accessed 11 Feb 2015), Feb 2015, https://gcc.gnu.org/gcc-5/changes.html.

[19] Google, "Issue 245: NaCl/x96 appears to leave return addresses unaligned when returning through the springboard," Web Page (accessed 28 Jul 2014), http://code.google.com/p/nativeclient/issues/detail?id=245.

[20] O. Horovitz, "Big loop integer protection," *Phrack*, no. 60, December 2002. [Online]. Available: http://phrack.org/issues/60/9.html#article

[21] *Intel 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 3A, 3B, and 3C*, 325462nd ed., Intel Corporation, Aug 2012, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf.

[22] "IntSafe," Web Page (accessed 14 May 2014), http://msdn.microsoft.com/en-us/library/windows/desktop/ff521693.

[23] ISO, *ISO/IEC 9899:2011 - Information Technology — Programming Languages – C*, 2012.

[24] D. Keaton, T. Plum, R. Seacord, D. Svoboda, A. Volkovitsk, and T. Wilson, "As-if infinitely ranged integer model," Carnegie Mellon Software Engineering Institute, Tech. Rep. CMU/SEI-2009-TN-023, July 2009, http://www.sei.cmu.edu/reports/09tn023.pdf.

[25] P. Mell, K. Scarfone, and S. Romanosky, "A complete guide to the common vulnerability scoring system version 2.0," Web Page (accessed 28 Jul 2014), http://www.first.org/cvss/cvss-guide.html.

[26] T. Mertes, “C as intermediate language, signed integer overflow and -ftrapv,” Web Page (accessed 24 Jul 2015), Jul 2014, https://gcc.gnu.org/ml/gcc/2014-07/msg00251.html.

[27] *MIPS Architecture For Programmers Volume II-A: The MIPS32 Instruction Set*, 5th ed., MIPS Technologies Inc., Sep 2013.

[28] D. Molnar, X. C. Li, and D. A. Wagner, “Dynamic test generation to find integer bugs in x86 binary Linux programs,” in *Proceedings of the 18th USENIX Security Symposium*, Aug 2009, pp. 67–82.

[29] Y. Moy, N. Bjorner, and D. Sielaff, “Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis,” Microsoft Research, Tech. Rep. MSR-TR-2009-57, May 2009.

[30] NIST, “National vulnerability database,” Web Page (accessed 28 Jul 2014), http://nvd.nist.gov.

[31] “Php: Integers,” Web Page (accessed 25 May, 2014, http://php.net/manual/en/language.types.integer.php.

[32] “Built-in types - python 3.5.0 documentation,” Web Page (accessed 25 May, 2014), https://docs.python.org/3.5/library/stdtypes.html.

[33] J. Regehr, “A guide to undefined behaviour in C and C++, part 1,” Web Page (accessed 25 May, 2014), University of Utah, Jul 2010, http://blog.regehr.org/archives/213.

[34] ——, “A guide to undefined behaviour in C and C++, part 2,” Web Page (accessed 25 May, 2014), University of Utah, Jul 2010, http://blog.regehr.org/archives/226.

[35] ——, “We need hardware traps for integer overflow,” Web Page (accessed 28 May, 2014), May 2014, http://blog.regehr.org/archives/1154.

[36] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. J. Beebee, “Enhancing server availability and security through failure-oblivious computing,” in *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation*, Dec 2004, pp. 303–316.

[37] “SafeInt,” Web Page (accessed 14 May 2014), http://safeint.codeplex.com.

[38] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *Proc. 11th ACM Conference on Computer and Communications Security*, Oct 2004, pp. 298–307.

[39] “Valgrind,” Web Page, http://valgrind.org.

[40] F. von Leitner, “Catching integer overflows in c,” Web Page (accessed 9 Feb 2015), Jan 2007, http://www.fefe.de/intof.html.

[41] T. Wang, T. Wei, Z. Lin, and W. Zou, “IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution,” in *Proceedings of the 16th Annual Network & Distributed System Security Symposium*, Feb 2009.

[42] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, “Improving integer security for systems with KINT,” in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, Oct 2012, pp. 163–177.

[43] R. Wojtczuk, “UQBTng: a tool capable of automatically finding integer overflows in Win32 binaries,” in *Proceedings of the 22nd Chaos Communication Congress*, Dec 2005.

[44] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, “Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time,” in *Proceedings of the 15th European Conference on Research in Computer Security*, Sep 2010, pp. 71–86.