Carnegie Mellon University
Research Showcase @ CMU

Department of Electrical and Computer Engineering

Carnegie Institute of Technology

2007

RICH: Automatically Protecting Against Integer-Based Vulnerabilities

David Brumley
Carnegie Mellon University

Tzi-cker Chiueh
SUNY Stony Brook

Robert Johnson
SUNY Stony Brook

Huijia Lin
Cornell University

Dawn Song
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/ece

Part of the Electrical and Computer Engineering Commons

Recommended Citation
.

This Conference Proceeding is brought to you for free and open access by the Carnegie Institute of Technology at Research Showcase @ CMU. It has been accepted for inclusion in Department of Electrical and Computer Engineering by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

RICH: Automatically Protecting Against Integer-Based Vulnerabilities

David Brumley, Tzi-cker Chiueh, Robert Johnson
dbrumley@cs.cmu.edu, chiueh@cs.sunysb.edu, rtjohnso@cs.sunysb.edu
Huijia Lin, Dawn Song
huijia@cs.cornell.edu, dawnsong@ece.cmu.edu

Abstract

We present the design and implementation of RICH (Run-time Integer CHecking), a tool for efficiently detecting integer-based attacks against C programs at run time. C integer bugs, a popular avenue of attack and frequent programming error [1–15], occur when a variable value goes out of the range of the machine word used to materialize it, e.g. when assigning a large 32-bit `int` *to a 16-bit* `short`*. We show that safe and unsafe integer operations in C can be captured by well-known sub-typing theory. The RICH compiler extension compiles C programs to object code that monitors its own execution to detect integer-based attacks. We implemented RICH as an extension to the GCC compiler and tested it on several network servers and UNIX utilities. Despite the ubiquity of integer operations, the performance overhead of RICH is very low, averaging about 5%. RICH found two new integer bugs and caught all but one of the previously known bugs we tested. These results show that RICH is a useful and lightweight software testing tool and run-time defense mechanism. RICH may generate false positives when programmers use integer overflows deliberately, and it can miss some integer bugs because it does not model certain C features.*

1 Introduction

C integer bugs are an underestimated source of vulnerabilities [1–15]. We present an evaluation of C integer vulnerabilities and the first automatic tool for efficiently and accurately detecting and preventing a large class of integer-based attacks against C programs. We have developed a static program transformation tool, called RICH, that takes as input any C program and outputs object code that monitors its own execution to detect integer overflows, underflows, and other bugs. Our experiments found that programs typically have hundreds to thousands of potential integer safety violations where bits may be lost or misinterpreted. Despite the ubiquity of integer operations, the run-time performance penalty of RICH is very low, averaging less than 5%. This result shows that it is practical to automatically transform C programs so that they are hardened against integer-based attacks.

Integer bugs appear because programmers do not anticipate the semantics of C operations. The C99 standard [16] defines about a dozen rules governing how integer types can be cast or promoted. The standard allows several common cases, such as many types of down-casting, to be compiler-implementation specific. In addition, the written rules are not accompanied by an unambiguous set of formal rules, making it difficult for a programmer to verify that he understands C99 correctly. Integer bugs are not exclusive to C. Similar languages such as C++, and even type-safe languages such as Java and OCaml, do not raise exceptions on some types of integer errors. However, for the rest of this paper, we focus on C.

The first step is to formally define the semantics of integer operations in C so that we may detect integer bugs correctly. One choice is to write out formally exactly what is specified by the C99 standard. This would lead to a formal specification of what many programmers already do not understand. In addition, it would leave gaping holes in important scenarios that the C99 standard defines as implementation specific.

Instead, our formal semantics of C integer operations captures the underlying idea that smaller precision types represent subsets of larger precision types with the same sign. Sub-typing theory is a natural way to express this relationship, e.g., `int8_t` is a subtype of `int16_t`, written `int8_t` $<:$ `int16_t`, because the values of `int8_t` $\subseteq$ `int16_t`. This approach mimics the approach in safer languages such as Ada, which allow users to create sub-types that qualify primitive types [17].

With the formal semantics in hand, we statically type-check the C program for safety. The purpose of static type checking is to decide whether a program will be safe to execute. At a high level, safety means that meaningful integer bits will not be lost or misinterpreted. Static type-checking is conservative, meaning that if a program fails type-checking it *potentially* uses an integer in a way that may lead to data loss or misinterpreted during computation.

Since C is unsafe, it should be of no surprise that many C programs do not type-check. At this point, there are several options: a) deem the program invalid and wait for it to be fixed, b) develop a more elaborate typing scheme that can better distinguish potential from real problems, c) automat-

ically promote integers such that integer bugs are impossible, or d) insert run-time checks that dynamically check safety. We found a typical program will have thousands of potential unsafe assignments, and if we count overflows, one potential error per 23 lines of code. Thus option (a) would require immense developer effort in order to fix all potential problems. Option (b) is plausible, but often very expensive since such analysis typically reasons about run-time behavior. For example, dependent typing [18] (where types can depend upon values) is one way of implementing (b), but is typically more expensive to check, and would seem to require the programmer to manually supply invariants and/or would be undecidable. Option (c), auto-promotion, is interesting since it eliminates most integer bugs, and is the approach taken by several languages such as SmallTalk and Scheme. Such an approach requires the ability to perform arbitrary precision arithmetic. This is not practical for C, since the run-time data layout assumes fixed-size integers. Furthermore, our experiments show that programmers rarely want arbitrary-precision integer semantics, so it does not make sense to make that the language default. Finally, arbitrary-precision integers may result in unpredictable performance. That leaves us with option (d): check all potentially unsafe operations at run-time.

Our choice to check all potential unsafe integer operations is a common way of back-porting type safety into an otherwise unsafe language. For example, the typing rule for arrays is unsafe in Java [19]. As a result, in Java each array access is potentially checked at run-time. In our setting, we check each potentially unsafe integer operation. The main trick is to make integer checks efficient. Though difficult to verify with language designers directly, it is commonly posted on the web that the reason some otherwise type-safe languages such as OCaml omit integer checks on base integer types is they believe them to be too expensive. We show that despite the number of potentially unsafe operations, we can implement checks such that they have very little overhead for C programs.

We implement the checks in two phases. At compile time, RICH instruments the target program with run-time checks of all unsafe integer operations. The current prototype instruments any C operation that could trigger an integer bug and our experiments show that the performance overhead of this instrumentation is very low. At run time, the inserted instrumentation checks each integer operation. When a check detects an integer error, it generates a warning and optionally terminates the program.

Some deliberate integer operations look like integer bugs, making it a challenge to perfectly detect integer-based attacks at run time. For example, programmers may use an integer overflow to perform a reduction modulo 2^{32} "for free". Like all overflows, the C source code contains no indication from the programmer that this overflow is intentional. Since the run-time detector cannot distinguish intentional overflows from attacks, the current prototype implementation generates warnings instead of aborting the program after detecting an integer error. Future versions of RICH could support programmer annotations to indicate intentional overflow sites in the program source code, enabling the run-time detector to abort the program or take other preventative measures whenever it detects an integer-based attack. Experiments applying RICH to real programs show that intentional overflows are quite rare, so annotating them would not be a burden on programmers.

We implemented RICH and measured its performance on several UNIX servers and utilities. Compiling applications with RICH required no source modifications – just a rebuild with our tool. For network servers like Apache and Samba, the performance overhead of RICH is extremely small, about 5%, even when the server is not I/O bound. False positives from RICH-protected programs revealed 32 intentional overflow sites in the 5 programs we benchmarked, demonstrating that intentional overflows are rare. RICH found 2 previously unknown integer bugs in Samba and ProFTPd, and detected 6 of the 7 previously known integer vulnerabilities we tested, missing one vulnerability because the program transformation models C pointers unsoundly. These experiments show that RICH is a useful and lightweight software testing tool and run-time defense mechanism that is backwards-compatible, efficient, effective, and easy to deploy.

We have also performed a comprehensive study of known integer bugs and exploits. Integer bugs fall into four broad classes: overflows[1], underflows, truncations, and signedness errors. Integer bugs are usually exploited indirectly, e.g., triggering an integer bug enables an attacker to corrupt other areas of memory, gaining control of an application. The survey results highlight the creativity of system hackers, as the surveyed exploits contained over a half-dozen different exploit strategies, all built upon minor-looking integer bugs. Even though integer bugs are often used to build a buffer overflow or other memory corruption attack, integer bugs are not just a special case of memory corruption bugs. If all programs were made memory safe, attackers would still find ways to exploit integer bugs.

Contributions. This paper presents an automatic approach for efficiently protecting against a large class of integer bugs, including overflows/underflows, truncation bugs, and sign conversion bugs in C programs. Specifically, we:

- Survey 195 known integer vulnerabilities and categorize them as overflows, underflows, truncation bugs, or sign conversion bugs.
- Provide formal semantics for safe C integer operations. We derive formal typing rules similar to those found in languages without integer bugs such as Ada. Our semantics replace the cumbersome and un-intuitive C99 specifications with a few simple sub-typing rules. In addition, since C is not type-safe, we also supply C-specific rewrite rules that rewrite any violation of the type-safety rules as a dynamic safety check.
- Implement a prototype called RICH (Run-time Integer CHecks) to evaluate our approach and techniques.

[1] "Integer overflow" is commonly used to describe integer overflows specifically and integer bugs in general.

- Demonstrate through experiments that potentially unsafe integer operations are rampant in source code.
- Show how to implement our dynamic checks with low overhead. In particular, although thousands of unsafe operations may be found and require checking, the average performance overhead is less than 3.7%.

Outline. We first describe several real-world integer bugs and outline exploits against these bugs in Section 2. In Section 3, we precisely define the security goal of our transformation and the theoretical foundations of our program transformation. Section 4 describes our tool RICH and Section 5 presents benchmark results. We discuss related work in Section 6 and draw concluding lessons in Section 7.

2 Integer Vulnerabilities

In this section, we first describe the different types of integer vulnerabilities in C programs. We also present a study we performed of 195 Common Vulnerability and Exploit (CVE) [20] candidate integer vulnerabilities. This study shows each type of integer vulnerability is common in source code. We then describe how integer vulnerabilities can be exploited to gain control of a program.

2.1 Integer Vulnerability Categories

Each integer type in C has a fixed minimum and maximum value that depends on the type's machine representation (e.g., two's complement vs. one's complement), whether the type is signed or unsigned (called "signedness"), and the type's width (e.g., 16-bits vs. 32-bits). At a high level, integer vulnerabilities arise because the programmer does not take into account the maximum and minimum values. Integer vulnerabilities can be divided into four categories: overflows, underflows, truncations, and sign conversion errors. Our study of 195 CVE known integer vulnerabilities indicates vulnerabilities from all categories are prevalent in source code.

Overflow. An integer overflow occurs at run-time when the result of an integer expression exceeds the maximum value for its respective type. For example, the product of two unsigned 8-bit integers may require up to 16-bits to represent, e.g., $2^8 - 1 * 2^8 - 1 = 65025$, which cannot be accurately represented when assigned to an 8-bit type. Officially, the C99 standard specifies that a "computation involving unsigned operands can never overflow" because the result can be reduced modulo the result type's width (page 34, [16]) (*signed* overflow is considered undefined behavior, thus implementation specific). However, overflows are currently the most common integer vulnerability, accounting for 148 of the 207 CVE vulnerabilities in our survey, indicating many programmers certainly do not understand or anticipate the C99 semantics.

Figure 1(i) shows a typical overflow vulnerability in GOCR [1], an optical character recognition program for processing images. An attacker can exploit the program by providing large integer values to the `inpam.width` and `inpam.height` fields. The product of these values used in the call to `malloc` will overflow, resulting in an erroneously small allocation. The small allocation allows the exploit to write out-of-bounds through `p`→`p`, shown on the last highlighted line.

Underflow. An integer underflow occurs at run-time when the result of an integer expression is smaller than its minimum value, thus "wrapping" to the maximum integer for the type. For example, subtracting $0 - 1$ and storing the result in an unsigned 16-bit integer will result in a value of $2^{16} - 1$, not -1. Since underflows normally occur only with subtraction, they are rarer than overflows, with only 10 occurrences in our survey. Figure 1(ii) shows a typical underflow vulnerability which occurs in Netscape versions 3.0-4.73 [3]. An attacker can specify the `len` field as 1, resulting in underflow in the expression `len-2`, thus assigning a large value to `size`. The following call to `malloc` would allocate 0 bytes (due to overflow in the expression `size+1`), allowing the attacker to overwrite memory on the subsequent `memcpy`.

Signedness Error. A signedness error occurs when a signed integer is interpreted as unsigned, or vice-versa. In twos-complement representation, such conversions cause the sign bit to be interpreted as the most significant bit (MSB) or conversely, hence -1 and $2^{32} - 1$ are misinterpreted to each other on 32-bit machines. 44 of the 195 CVE vulnerabilities in our survey are signedness errors. Figure 1 (iii) shows a signedness error from the XDR (eXternal Data Representation, used by Sun RPC and NFS) routines in Linux kernel 2.4.21[2] in which the signed `int size` is initialized directly from unsigned, attacker-controlled XDR data, `*p`. A negative `size` value bypasses the signed upper bound check but is interpreted as a very large positive number by the `memcpy` function, whose size argument is unsigned, resulting in an instant kernel panic.

Truncation. Assigning an integer with a larger width to a smaller width results in integer truncation. For example, casting an `int` to a `short` discards the leading bits of the `int` value, resulting in potential information loss. Figure 1(iv) shows a truncation vulnerability from the SSH CRC-32 Compensation Attack Detector [4]. The local variable `n` is only 16-bits long, so the assignment `n = l` can cause a truncation. By sending a very large SSH protocol packet, an attacker can force this truncation to occur, causing the `xmalloc` call on the next line to allocate too little space. The code that initializes the allocated space a few lines later will corrupt SSH's memory, leading to an attack.

2.2 Exploiting Integer Bugs

Integer bugs differ from other classes of exploits because they are usually exploited indirectly. Typical exploits include 1) Arbitrary code execution such as when an integer vulnerability results in insufficient memory allocation, which is subsequently exploited by buffer overflows, heap overflows, overwrite attacks, etc; 2) Denial of Service (DoS) attacks where the exploit causes excessive memory

```
struct pixmap {
    unsigned char *p;
    int x;          /* xsize */
    int y;          /* ysize */
    int bpp;
};
typedef struct pixmap pix;
......
void readpgm(char *name, pix * p) {
    /* read pgm */...
    pnm_readpaminit(fp, &inpam);
    p->x=inpam.width;
    p->y=inpam.height;
    if(!(p->p=(char *)malloc(p->x*p->y)))
        F1("Error at malloc");
    for(i=0; i<inpam.height; i++){
        pnm_readpamrow(&inpam, tuplerow);
        for(j = 0; j<inpam.width; j++)
            p->p[i*inpam.width+j]=sample;
    }
}
```

(i)

```
void getComm(unsigned int len, char *src){
    unsigned int size;
    size = len - 2;
    char *comm = (char *)malloc(size + 1);
    memcpy(comm, src, size);
    return;
}
```

(ii)

```
static inline u32 *
decode_fh(u32 *p, struct svc_fh *fhp) {
    int size;
    fh_init(fhp, NFS3_FHSIZE);
    size = ntohl(*p++);
    if (size > NFS3_FHSIZE)
        return NULL;
    memcpy(&fhp->fh_handle.fh_base,
            p,size);
    fhp->fh_handle.fh_size = size;
    return p + XDR_QUADLEN(size);
}
```

(iii)

```
int detect_attack(u_char *buf, int len, u_char *IV){
    static word16  *h = (word16 *) NULL;
    static word16   n = HASH_MIN_ENTRIES;
    register word32 i, j;
    word32 l;
    ...
    for(l=n; l<HASH_FACTOR(len/BSIZE); l=l<<2);
    if (h == NULL) {
        debug("Install crc attack detector.");
        n = l;
        h = (word16 *) xmalloc(n*sizeof(word16));
    } else
    for (c=buf, j=0; c<(buf+len); c+=BSIZE, j++){
        for (i = HASH(c) & (n - 1); h[i] != UNUSED;
            i = (i + 1) & (n - 1)) ...;
        h[i] = j;
    }
}
```

(iv)

Figure 1. (i) GOCR PNM image size integer overflow vulnerability. (ii) Netscape JPEG comment length integer underflow vulnerability. (iii) Linux kernel XDR integer signedness errors. (iv) SSH CRC-32 Compensation Attack Detector integer truncation vulnerability. In each figure, the integer bug is highlighted with a pink background; the resulting exploit is highlighted in blue.

allocation or infinite loops; 3) Array index attacks where a vulnerable integer is used as an array index, so that attackers can accurately overwrite arbitrary byte in memory; 4) Bypassing sanitization attacks, such as bypassing an upper bounds check that does not take into account unexpected negative integer values; and 5) Logic errors, for example as in NetBSD where an integer vulnerability allowed an attacker to manipulate a reference counter, causing the referenced object to be freed prematurely.

The security costs of integer bugs is severe: 125 of the 207 entries may lead to arbitrary code execution, 70 can cause denial of service attacks, 10 lead to privilege escalation, and 14 result in invalid memory accesses or memory exhaustion. 63 of the integer exploits are followed by buffer overflows; in 14 cases, they pollute the size arguments of memory allocation/manipulation functions; and in another 3 instances, they are exploited in conjunction with format string vulnerabilities.

Note solving buffer overflows, malloc errors, and format string bugs would still leave many integer vulnerabilities exploitable. Many vulnerabilities in our study can be exploited in more than one way, e.g., an integer vulnerability that can be abused to cause a buffer overflow or a denial of service attack. This indicates it is insufficient to fix a particular way of exploiting an integer vulnerability, as there may be alternate ways the vulnerability can be exploited to take control of the program. In addition, several of the vulnerabilities in our survey exploited application-specific logic errors, thus there are no likely application-independent quick-fixes. Previous vulnerability prevention mechanism such as StackGuard [21], CCured [22], etc. can prevent some attacks, but they are *insufficient* to combat integer vulnerabilities.

3 Safe Integer Semantics

In this section, we present our approach for protecting against integer vulnerabilities. Our approach is motivated by type-safe languages that do not have integer vulnerabilities. We define type-safety rules for C integer operations and apply them to programs. When we find a violation of our typing rules, we insert a dynamic check which decides at run-time whether the (static) safety violation results in an integer violation.

In the following presentation, we assume that all implicit conversions and casts have been made explicit before these typing and rewrite rules are applied. We also assume that implicit arithmetic, such as the address computation in an array reference `a[i]`, has been made explicit.

3.1 Sub-typing Rules for Safe Integer Operations

Our approach adds type-safety to C integers by applying sub-typing theory to integer types. Table 1 summarizes our four typing rules T-UNSIGNED, T-SIGNED, T-US (unsigned to signed), and T-UPCAST. Due to space, we provide in Appendix A the full set of rules, and a more detailed exposition in our companion technical report [23]. Our integer sub-typing rules are similar to those found in type-safe languages [19], such as Ada [17]. The intuition in our scenario is to read the sub-typing relationship "$<:$" as "$\subseteq$", i.e., if integer type `int`a`_t` $<:$ `int`b`_t`, then the values of `int`a`_t` $\subseteq$ `int`b`_t`. For instance, T-US specifies that if $n < m$, then `uint`n`_t` $<:$ `int`m`_t` because any n-bit *unsigned* integer can be represented as an $n+1$-bit (or greater) *signed* integer. Similarly, T-UPCAST specifies upcasts are always safe since a larger-width type can always represent a smaller-width type. Pointer arithmetic is treated as unsigned integer arithmetic, so basic pointer operations are also handled by our typing rules.

$$\frac{}{\text{uint8_t} <: \text{uint16_t} <: \text{uint32_t} <: \text{uint64_t} <: \mathbb{Z}}\ \text{T-UNSIGNED}$$

$$\frac{}{\text{int8_t} <: \text{int16_t} <: \text{int32_t} <: \text{int64_t} <: \mathbb{Z}}\ \text{T-SIGNED}$$

$$\frac{n < m}{\text{uint}n\text{_t} <: \text{int}m\text{_t}}\ \text{T-US} \quad \frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash (\tau)e : \tau}\ \text{(T-UPCAST)}$$

$$\frac{\Gamma \vdash e : \sigma \quad (\tau)e \quad \sigma \not<: \tau \quad \sigma, \tau <: \mathbb{Z} \quad e \rightsquigarrow e'}{(\tau)e \rightsquigarrow (\tau)\text{let } x : \sigma = e' \text{ in if } \tau_{\text{min}} \leq x \leq \tau_{\text{max}} \text{ then } x \text{ else error}}\ \text{R-UNSAFE}$$

$$\frac{\Gamma \vdash (e_1 \Box e_2) : \tau}{e_1 \Box e_2 \rightsquigarrow (\tau)\text{let } x : \mathbb{Z} = (\mathbb{Z})e_1 \Box (\mathbb{Z})e_2 \text{ in } \text{CHECK}_{\tau(\mathbb{Z})}(x)}\ \text{R-BINOP}_{\mathbb{Z}}$$

Table 1. Our rules for type safety (T-UNSIGNED, T-SIGNED, T-US, and T-UPCAST), and for rewriting potentially unsafe integer operations (R-UNSAFE and R-BINOP$_{\mathbb{Z}}$)

In our semantics, an integer expression is always safe iff it is well typed. For example, code `uint8_t b; uint16_t a = (uint16_t) b;` is safe because it is well-typed, as given by the typing derivation:

$$\frac{\Gamma \vdash \mathsf{b} : \text{uint8_t} \quad \overline{\text{uint8_t} <: \text{uint16_t}}\ \text{T-UNSIGNED}}{\Gamma \vdash (\text{uint16_t})\mathsf{b} : \text{uint16_t}}\ \text{T-UPCAST}$$

3.2 Rewriting Potentially Unsafe Truncations and Sign Conversions

Truncations (down-casts) and sign conversions are not within the type system because they can lead to data loss, and are therefore potentially unsafe. Thus, we add new rules that cover potentially unsafe operations. We rewrite potentially unsafe casts as run-time safety checks on the operands, as shown by rule R-UNSAFE in Table 1. Note these rewriting rules are not found in (static) type-safe languages. Statically typed languages respect a phase distinction between compile-time and run-time processing. Our rewriting rules introduce run-time checks that ensure type safety, i.e., they make C integer operations dynamically type safe.

R-UNSAFE states that a potentially unsafe cast $(\tau)e : \sigma$, where e has been rewritten to some other expression e', is rewritten statically to another cast where e' is evaluated to a value x. The value x is checked at run-time for data loss (i.e., if $\tau_{\text{min}} \leq x \leq \tau_{\text{max}}$ is false); if so, an `error` is raised. In our implementation, `error` can be defined to abort the program, or simply raise a warning.

For example, consider the potentially unsafe down-cast: `uint32_t b; uint16_t a =(uint16_t) b;` Here $\sigma =$ `uint32_t` and $\tau =$ `uint16_t`. Since `uint32_t` $\not<:$ `uint16_t`, the rewriting rule R-UNSAFE applies. Figure 2 shows how R-UNSAFE is instantiated in this example.

Thus, the rewritten C statement is:

```
if(b > 2^16 − 1 || b < −2^16) then error();
a = b;
```

3.3 Rewriting Potential Overflow and Underflow Operations

In C, *addition*, *subtraction*, *multiplication*, *negation*, and *division* may all result in overflow or underflow. The first three are self-explanatory, while negation and division overflow in a subtle way. Signed integer types with twos-complement representation have asymmetric ranges, *i.e.* $[-2^{n-1}, 2^{n-1} - 1]$. When -2^{n-1} is negated or divided by -1, the result overflows and wraps back to -2^{n-1} itself.

We rewrite via R-BINOP$_{\mathbb{Z}}$ any arithmetic that may result in overflow/underflow to be performed in a virtual type $\mathbb{Z}$ which has infinite width. [2] Arithmetic in $\mathbb{Z}$ cannot overflow/underflow. Since $\mathbb{Z}$ is not a sub-type of any other type, we apply R-UNSAFE to the result. If there would have been an underflow/overflow without the rewrite, R-UNSAFE will raise a warning, i.e., the result requires more bits to represent than we have in specified type.

In practice, we do not implement arbitrary precision arithmetic for $\mathbb{Z}$. Instead, we simply up-cast arithmetic to an appropriate type large enough to represent the result. [3] If the architecture does not support a large enough type, the arithmetic is performed in software. For x86 with normal C types, casts to software are quite rare and usually only happen with 64-bit integers. Thus, we avoid the problem of always upcasting to arbitrary precision by only upcasting to the next higher precision, and only potentially using software for 64-bit integers. This is different than SmallTalk and Scheme as there is no arbitrary precision type.

For example, our approach rewrites `int8_t a, b, res; res= a+b;` as:

```
int8_t a, b,res;   int16_t t16;
t16 = (int16_t) a + b;
if(t16 > 2^7 − 1 || t16 < −2^7) error();
res = (int8_t) t16;
```

Note we only check C operations that may result in overflow/underflow. For example, bit-wise and logical ands, ors,

[2] For brevity, we omit the corresponding unary rule.

[3] Theoretically in some cases we may not need up-casting at all by employing various mathematical tricks for the check.

$$\frac{\Gamma \vdash b : \text{uint32_t} \quad \text{uint32_t} \not\leq : \text{uint16_t} \quad b \rightsquigarrow b'}{(\text{uint16_t})b \rightsquigarrow (\text{uint16_t}) \text{ let } x : \text{uint32_t} = b \text{ in if } -2^{16} \leq x \leq 2^{16} - 1 \text{ then } x \text{ else error}} \text{ R-UNSAFE}$$

Figure 2. An instance of R-UNSAFE for the program `uint32_t b; uint16_t a =(uint16_t) b;`

and negations do not correspond to any arithmetic operation on integers, thus are not checked. If a programmer performs arithmetic using these operations we do not check the result. Adding such checks is straight-forward, but would likely break many programs. Thus, we do not implement this feature, as it would only be useful in a limited number of cases.

3.4 Limitations

Since C specifically allows potentially unsafe behavior, it is impossible to ensure safety in a way backward compatible with all programs. As a result, there are two main limitations to our approach: we do not handle certain unsafe pointer aliasing relationships, and programs that specifically rely on certain potentially unsafe C99 features may break.

Potentially Unsafe Pointer Aliasing. We currently do not check potentially unsafe pointer aliasing relationships when two pointers of different types alias the same memory cell. For example, if data is written to memory through a `uint16_t *` pointer, a subsequent read through an `uint8_t *` pointer will go unchecked. Checking this sort of potential error would require maintaining run-time meta-data, *a la* CCured[22]. Alternatively, we could use an alias analysis to find all the pointers that may alias each other, and require that writes through those pointers can be safely read by all other pointers in the alias set. This would essentially require the writes to fall within the smallest common type of the aliased pointers. Our survey also shows that integer vulnerabilities through pointer aliasing is very rare.

Unions create a similar aliasing situation as pointers. For similar reasons, we ignore this aliasing, effectively treating unions like structures.

Deliberate Use of Potentially Unsafe C99 Features. Since C is not type-safe, type-safe semantics in general do not directly correspond to C99 semantics. In fact, C99's semantics are much more complicated than our type-safety approach: C99 requires about a dozen rules to specify integer coercions, while we require only 2 rules. [4] However, some programs are deliberately written to take advantage of potentially unsafe C features. For example, a program may be written to take advantage of C99's rule where overflow is reduced modulo the destination register size, e.g., `(int32_t)a + (int32_b) b;` is written as shorthand for `((int32_t)a + (int32_b) b)` $\bmod 2^{32}$`;`. Our approach will raise an `error` when such statements are executed.

Our experiments indicate a typical program may have *thousands* of potentially unsafe operations, while only a few of those operations rely on otherwise unsafe features. Overall, we take the approach that it is better to check the thousands of potentially unsafe vulnerabilities and have the programmer annotate or rewrite the few seemingly unsafe operations. Our study of 195 CVE vulnerabilities suggests this course is the prudent choice. Thus, RICH strikes a balance between automatic protection, backwards compatibility, and common programming practices. Our experiments indicate this balance seems appropriate for most programs.

False Positives and False Negatives. At a high level, because of the above two limitations, our transformations may result in false positives and false negatives. A false positive occurs when we raise `error` when the programmer deliberately uses potentially unsafe features of C. A false negative occurs when we do not raise an error due to pointer aliasing.

4 Implementation

We implemented a prototype of our strong integer typing approach in a tool called RICH. RICH has two implementations: 1) as a platform-specific compiler extension to GCC 3.4.1, and 2) as a source-to-source transformation. The former approach demonstrates platform-specific optimizations, while the latter is platform independent.

Our GCC 3.4.1 version of RICH checks the GCC intermediate representation (IR) of the source code during the code-generation phase, and instruments any potentially unsafe integer operations. By working directly in the compiler, we can take advantage of architecture-specific instructions to implement R-UNSAFE and R-BINOP$_{\mathbb{Z}}$. Specifically, we use the IA-32 `jo, jc, js`, instructions, which jump to a given target if the last arithmetic operation resulted in an overflow, carry in the high-bit, or result with the sign-bit set. Figure 3 shows example checks in our GCC implementation for several common arithmetic operations.

The downside of our GCC implementation is that the checks may apply to compiler-generated, harmless overflows. For example, the compiler's constant propagation pass may negate unsigned constants. If the GCC module inserts checks for these operations, the program will fail. We solve this problem by disabling optimizations that can introduce benign overflows. With further work, we could likely work around this problem.

We have also implemented RICH as a platform-independent source-to-source transformation that works with any compiler [23]. The source-to-source transformation is written as a CIL plug-in [24, 25], a C analysis framework written in OCaml. CIL reads in the source code, performs several semantic-preserving simplifications, and then produces a typed intermediate representation (IR). Our

[4] We believe the simplicity alone is a strong reason to support our approach: complicated systems are usually the most error prone.

```
add'⟨32,s⟩ ← ⟨32,s⟩×⟨32,s⟩ op1, op2
  := add op1, op2
     jo VH
add'⟨32,u⟩ ← ⟨32,u⟩×⟨32,u⟩ op1, op2
  := add op1, op2
     jc VH
sub'⟨32,s⟩ ← ⟨32,u⟩×⟨32,u⟩ op1, op2
  := sub op1, op2
     jc .L1
     js VH
     .L1
     jns VH
lo'⟨16,s⟩ ← ⟨32,u⟩ source, dest
  := mov source, %eax
     shrl %eax, 15
     and %eax, %eax
     jne VH
     lo source, dest
```

```
cv'⟨32,v̄⟩ ← ⟨32,v⟩ operand
  := mov operand, %eax
     and %eax, %eax
     js VH
     cv operand
div'⟨32,s⟩ ← ⟨32,s⟩×⟨32,s⟩ top, bottom
  := xor %ebx, %ebx
     mov top, %eax
     xor 0x8000000,%eax
     je .L1
     mov bottom, %eax
     xor 0xfffffff, %eax
     je .L1
     mov 1, %ebx
     L1:
     cmp 0, %ebx
     jne VH
     div dividend, divisor
```

Figure 3. Instrumented operators and algorithm for de-reference type casts. The instructions `jo`, `jc`, and `js` jump to the specified target if the last arithmetic operation yielded an overflow, a carry off the high bit, or result with MSB 1, respectively. The `jno`, `jnc`, and `jns` instructions negate these tests. The overflow handler is called `VH`. The cv operation converts between signed and unsigned, and lo takes the low-order bits of its operand.

CIL implementation then performs the transformation described in the previous section on the IR, which is then "unparsed" by CIL and written to a file. The resulting file is C source code containing the necessary checks, which can then be compiled with any standard C compiler. The primary advantage of this approach is that the compiler can perform all its optimizations. The disadvantage is that the source-to-source translation cannot directly take advantage of architecture-specific optimizations such as using the CPU status bits to perform efficient overflow checks.

Note our CIL-based solution does not implement overflow/underflow checks (R-BINOP$_{\mathbb{Z}}$), only signedness and truncation checks (R-UNSAFE). For that reason, the benchmark results given in Section 5 all use the GCC implementation of RICH. Experiments with the CIL implementation show similar overheads.

5 Evaluation

We evaluated RICH with several server and utility applications, several of which contain real vulnerabilities. Overall, we found programmers typically do not write safe code. However, the overhead of protecting against exploits in the unsafe code is quite low, averaging less than 3.7%.

The evaluation was performed on an Intel Pentium M 1.6MHz machine with 512MB memory and Linux 2.6.9 kernel. In our experiments, we defined `error` to log each violation of our dynamic checks. Our benchmark suite includes Apache 2.2.0, Samba 2.2.7a, ProFTPD 1.2.10, and gzip 1.2.4.

5.1 Check Density

We first measured how many checks our approach inserts into our test-suite programs. Table 2 shows, for each benchmark program, the number of lines of code in that program and the number of checks inserted by the RICH compiler extension. The numbers also give a rough idea how many potentially unsafe integer operations exist in code, though since our analysis is conservative it is definitely an overcount.

Unsafe operations can generally be divided into two categories: potential runtime errors due to overflow/underflow, and static casting/truncation errors. We found hundreds to thousands of static type errors, indicating programmers ignore safety issues. While many of these errors are not exploitable, they are dangerous enough that type-safe languages would generally reject a program containing even one error. Adding in overflow/underflow raises the number to about one potential problem every 23 lines of code.

This shows that programmers use potentially-overflowing or otherwise buggy operations all the time, so the chance that at least one of them is a real, exploitable bug is significant. This highlights the need for a defense mechanism like RICH.

5.2 Performance Overhead

Although the number of inserted checks can be quite high, our experiments indicate the overhead is quite low, averaging less than 3.7%. Table 3 shows the performance overhead of RICH-enabled applications relative to the uninstrumented versions (both compiled with the same flags). The Apache web server was tested with the web benchmark *ab*, distributed with Apache, configured to generate 20,000 requests to a local Apache server and to use a concurrency level of 100 requests. For ProFTPD, we used an open source FTP benchmark tool, *dkftpbench*, that runs a 10-second simulation of 10 users repeatedly logging in, transferring a file, and logging out. For Samba 2.2.7a, we first used Bonnie++, a standard benchmark for hard drive and file system assessment. Since the Bonnie++ benchmark

Program	Size(KLOC)	Signed OF/UF	Unsigned OF/UF	Trunc.	Sign Conv.	All
Apache Httpd 2.2.0	101	3958	2454	2279	642	9315
ProFTPD 1.2.10	48	938	483	478	269	2168
Samba 2.2.7a	189	7399	7597	4603	2195	21794
gzip 1.2.4	7	572	195	421	138	1326

Table 2. Number of checks inserted.

is largely I/O bound, and our checks are CPU bound, we wrote a script that performed several simple file-system operations on a small set of files that could be cached in main memory. We primed the cache with 5 runs of the script and then timed it on 645 subsequent iterations.

To stress test RICH with a CPU-bound application, we measured the performance overhead for *gzip*, a compression utility. At first, gzip showed over a 300% slowdown while decompressing the archive `linux-2.6.15.tar.gz`. A closer examination revealed that the integer error handler was being triggered repeatedly by two deliberate integer violations in an inner loop of the gzip inflation code. Both violations intentionally make use of overflow to perform reduction modulo 2^{32}. If the prototype implementation supported annotations, a programmer could annotate these two intentional overflows to eliminate both the check for overflow and the consequent `error` call, eliminating the overhead. To estimate the speedup this would yield, we recompiled gzip with the RICH checks in place but with `error` defined as a no-op . This cut the overhead to only 1.77%.

Comparison with GCC's built-in protection. We also compared the performance of RICH with GCC's `-ftrapv` option, as shown in the last column of Table 3. The `-ftrapv` option only checks for signed overflows in explicit arithmetic operations, so it is much less comprehensive than RICH and, as our experiments show, has much higher overhead. Furthermore, `-ftrapv` does not allow a programmer to specify an action when overflow is detected; it always aborts. Thus, we could not measure the `-ftrapv` overhead for Samba and Apache since they contain overflows. For ProFTPD, `-ftrapv` slightly outperformed RICH, but keep in mind that the protection it provides is much weaker. However, even for computation intensive programs like gzip, RICH greatly outperformed `-ftrapv`. This is because `-ftrapv` is implemented as a function call to a checking routine, while RICH takes advantage of IA32 instructions like `jo` to check for integer errors. Overall, although RICH protects against more than `-ftrapv`, it still achieves better performance by making those checks extremely efficient.

Analysis of Dynamic Check Overhead. The performance overhead in Table 3 is somewhat surprising, given the number of checks inserted. There are several reasons the overhead is so low. In RICH, the majority of integer checks are implemented as a single instruction, and do not require any extra loads or stores. Thus, the instructions can be pipe-lined, and given the excellent branch prediction in modern CPU's, executed essentially for free. On the other hand, GCC's signed overflow check is a function call, thus quite expensive. Overall, although we could perform additional optimizations to remove unnecessary checks (e.g., redundant check elimination), we found the current overhead so low there was no need.

5.3 False Positives and False Negatives

5.3.1 Analysis of False Positives

We measured the number of deliberate uses of unsafe C features, as discussed in Section 3.4. Table 4 shows our results. We measured 32 total deliberate uses, which we then broke down into the following categories:

Pseudo-random number generation. Pseudo-random number generators often use potentially unsafe C operations as short-cuts. Examples can be found in hash functions in Samba, Apache, and Pine, as well as in ProFTPD's memory scrubbing function.

Message encoding and decoding. Data-structures in network applications sometimes need to be serialized to messages to exchange status between client and server. During marshaling and de-marshaling, types are often cast from one type to another incompatible type, e.g., encoding a series of `uint16_t`'s as an unsigned character byte stream.

Integer as ID. When an integer is used as an ID, only the bit sequence matters. Signedness conversion errors which do not affect bit values are thus benign.

Mixed usage of signed and unsigned `char`. There are many instances of assignments between signed and unsigned `char` in Samba and Pine, which are essentially innocent as long as they are not interpreted incorrectly (note even benign cases can often result in portability bugs [26]).

Explicit casts. Programmers can use explicit casts for several reasons, including to reduce an integer modulo a power of two and to access subsets of an integer's bits as a bit-vector.

Table 5 shows a break-down for how many false positives each programming pattern caused in our test suite.

5.3.2 Analysis of False Negatives

We tested RICH on several vulnerabilities in real programs to verify that it can effectively prevent integer-based attacks. Some vulnerabilities actually consist of two integer bugs, like the Pine and Apache mod_auth_radius vulnerabilities. Table 6 shows the results of these experiments.

Integer exploits are fragile, so our instrumentation could potentially perturb the program layout sufficiently to ren-

Program	Benchmark	Overhead(RICH)	Overhead(-ftrapv)
Apache Httpd 2.2.0	ab	8.18%	N/A
ProFTPD 1.2.10	dkftpbench	3.59%	0.84%
Samba 2.2.7a	Bonnie++	1.21%	N/A
Samba 2.2.7a	script	3.59%	N/A
gzip 1.2.4	decompress linux-2.6.15.tar.gz	1.77%	48.46%

Table 3. Performance overhead of instrumented applications.

Program	Total No.	Overflow / Underflow	Sign. conversion	Truncation	Real Bug
Samba 2.2 7a	15	3	1	10	1
gzip 1.2.4	3	2	0	1	0
Apache 2.2.0	7	4	2	1	0
ProFTPD 1.2.10	3	0	0	2	1
Pine 4.55	6	2	2	2	0

Table 4. Technical break down of false positives.

der a known exploit inoperative. To get around this problem, we extracted the vulnerable code in each benchmark into a small, standalone program and then called that code with error-inducing input. As Table 6 shows, RICH caught all but one of the exploitable integer bugs. The only false negative occurs because the current RICH prototype does not support pointer alias analysis and therefore cannot catch implicit casting when a variable is accessed through pointers of different types. Table 6 also shows that despite its much higher run-time overhead, the GCC `-ftrapv` option is much less effective at catching integer exploits than RICH.

5.4 New Bugs

RICH uncovered two new integer vulnerabilities in the benchmark programs used in the performance tests. Samba 2.2 7a passes a pointer difference, `name-(*start)`, as the length argument to `strncpy` at `statcache.c:206`, but this value was negative in one of the performance benchmark runs, resulting in a signedness error when it is converted to an `unsigned int`. This causes `strncpy`'s size argument to become unusually large, leading to a possible buffer overflow. This bug was fixed in CVS as part of a general rewrite of `statcache.c` in March, 2003, but we could find no evidence in mailing lists or CVS logs that the developers were specifically trying to fix this bug with that rewrite.

ProFTPd 1.2.10 translates the fields of the UNIX `/etc/shadow` file into internal data-structures. One step of this conversion translates the password age fields of `/etc/shadow`, which are expressed in days, into seconds by multiplying by 86400. This multiplication overflowed in some of the benchmark runs. An attacker could potentially use this overflow to log in using an expired password. This bug still exists in current versions of ProFTPd.

6 Related Work

C/C++. Several authors have proposed replacing some or all integers in a program with safe integers implemented by a shared library that, like RICH, checks for integer errors at run time. The GCC run-time library provides functions that trap on signed arithmetic overflows, such as `__addvsi3`. However, the library only considers signed addition, subtraction, and multiplication, so it is not a comprehensive solution.

The Microsoft Visual C++ compiler and GCC consider different, but incomplete, aspects of integer security. The Visual C++ .NET 2003 compiler provides a compiler switch, RTCc, that generates run-time checks for truncation with data loss. The Visual C++ .NET 2005 compiler is able to catch overflows in the `::new` operator, when calculating the byte size of a non-character array. GNU GCC compilers since version 3.4 have provided the `-ftrapv` command-line flag that causes GCC to generate assembly code using its safe integer functions mentioned above. The protection is confined to signed arithmetic and does not include sensitive address calculations in array and component references. The Big Loop Integer Protection (BLIP) [27] compiler extension transforms programs to detect overly large counters. BLIP uses a fixed threshold, resulting in many false negatives and false positives.

David LeBlanc's SafeInt C++ template class [28] overrides almost all relevant operators, including division, negation, assignment and comparison. Michael Howard provides a small in-line assembly library [29] that checks for arithmetic overflows and underflows using exactly the overflow detecting instructions (`jo`, `jc`, etc.) as RICH. Howard has also released IntSafe [30], a safe integer arithmetic library of unsigned math functions and conversion routines optimized for performance and reportedly used in Windows Vista.

Arbitrary precision arithmetic packages, like the GNU Multiple Precision Arithmetic Library (GMP), can also help circumvent integer security problems. They support arbi-

Program	Total	PRNGs	I/O Marshaling	Mixed `Chars`	Integer ID	Explicit casts
Samba 2.2 7a	14	4	3	7	0	6
gzip 1.2.4	3	0	0	1	0	0
Apache 2.2.0	7	2	0	0	3	1
ProFTPD 1.2.10	2	2	0	0	0	0
Pine 4.55	6	2	1	2	1	3

Table 5. Breakdown of programming patterns which cause false positives.

Program	Vulnerability	Kind(s)	Caught(RICH)	Caught(-ftrapv)
Samba 2.2.7a	`statcache.c:206`	Sign err	Yes	No
Samba 2.2.7a	`replay_nttrans()` bug[7]	OF	Yes	No
Samba 2.2.7a	Directory ACL bug[12]	OF	Yes	No
ProFTPD 1.2.10	`mod_auth_unix.c:434`	OF	Yes	Yes
Pine 4.55	Header parsing bug[8]	UF, OF	Yes, Yes	Yes, No
Mailutil-0.6 Imap4d	`fet_io()` bug[13]	OF	Yes	No
PuTTY 0.53b	SFTP Client bug[11]	Sign Err	Yes	No
mod_auth_radius 1.5.7	RADIUS reply bug[10]	UF, Sign err	Yes, Yes	No, No
GNU Radius 1.2	SNMP DoS[9]	Sign Err	No	No

Table 6. Results of testing RICH with real application vulnerabilities. UF stands for underflow while OF for overflow. Sign Err is signedness conversion error

trarily large integers, making overflows and underflows impossible. As observed above, most programmers do not intend computations to overflow, so providing support for very large integers that should not arise in practice may be overkill in many scenarios.

Other Languages. We target C specifically. Some languages are safe with respect to integer operations, such as Lisp and Ada. In typical safe languages, the type-checking rules prevent down-casts. We use a sub-typing semantics, which is similar that found in Ada [17], as the basis for our type-checking rules. Overflow/underflow is generally handled in safe languages by either inserting run-time checks which raise exceptions, or automatically promoting integers to arbitrary-precision arithmetic, as discussed in Section 1. Interestingly, we say many type-safe languages, not all, because some otherwise type-safe languages do not protect against overflows, e.g., Java and OCaml. As a result, programs written in these languages may contain errors. Our analysis indicates that such checks can be implemented efficiently, and thus such languages should consider adopting run-time checks for overflow/underflow.

Our approach will raise a warning or throw an error when an unsafe downcast or overflow occurs at runtime. Runtime warnings/exceptions may pose a denial-of-service risk. There has been significant previous work in proving that a program will never have an unhandled error that may help address this problem. For example, SPARK/Ada can guarantee programs written in their framework are free of runtime errors such as overflows [31, 32]. Model checking the C program is another approach which could be used to prove runtime errors cannot happen, e.g., [35–38].

Our primary goal is to cheaply make existing C programs safe from exploits such as buffer overflows. Statically checking that the inserted run-time checks do not introduce a new denial-of-service attack is beyond the scope of this work.

There is work on safe variants of C, such as CCured [33] and Cyclone [34]. Our work differs from theirs as we are fixing a single problem in C without requiring any user intervention. CCured and Cyclone offer greater protection, but often require manual effort to convert a C program to the safe language variant.

Other techniques. ASTREE is a sound but incomplete static analyzer which can prove the absence of integer errors on a limited subset of C (without recursion or dynamic memory allocation) [39]. Another unsound and incomplete approach is extended static checking [40, 41]. However, at the end of the day, unsound or incomplete techniques offer no guarantees.

We could reduce the number of checks through additional analysis. Dataflow analysis [42, 43] could be used to reason about possible ranges of values and thus allow us to remove unneeded or duplicate checks. Some of the techniques developed by Redwine and Ramsey for porting applications to architectures with wider integer word sizes [44] could also be applied to optimize away unnecessary RICH checks. Initially, we expected that we may adopt some of these techniques to prune out unnecessary checks, but our performance numbers indicate this is unneeded.

Other more expressive type systems could be brought to bear [18, 45, 46]. Dependent types allow types to depend on values [18], and could be used to express integer safety. Dependent types are more powerful, but can also be more expensive to check. In particular, it seemed difficult to come up with a type system that was decidable yet more useful than our simple sub-typing system.

Since integer violations usually work in tandem with other attacks, techniques for preventing buffer overflows

and enforcing input sanitization help alleviate the damage of integer violations[22, 47, 48]. However, these can only be auxiliary solutions and cannot solve the integer security problem in general.

7 Conclusion

This paper surveys integer-based attacks and provides a theoretical framework to formally define and reason about integer errors soundly. The RICH compiler extension is, to the best of our knowledge, the first comprehensive, automatic integer error prevention tool for C. Experiments with real servers and UNIX utilities show that RICH is backward compatible, easy to use, efficient, and effective.

Acknowledgments

We thank Maohua Lu for helping with an early implementation of the RICH compiler extension. We also thank Joseph Slember for his help with this work in its early stages. Finally, we thank Neel Krishnaswami, Vyas Sekar, the anonymous reviewers, and Jonathan Shapiro for their thoughts, comments, and help on early revisions of this paper.

References

[1] "Gocr ReadPGM NetPBM remote client-side integer overflow vulnerability," CVE, April 2005. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-1141

[2] "Linux kernel (prior to 2.4.21) XDR routine interger signedness error," CVE, Jul 2003. [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0619

[3] "JPEG COM marker processing vulnerability in netscape browsers," Solar Designer, July 2000. [Online]. Available: http://www.openwall.com/advisories/OW-002-netscape-jpeg/

[4] "SSH CRC-32 compensation attack detector vulnerability," CVE, Feb 2001. [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144

[5] "Linux kernel do_brk() vulnerablility," CVE, Dec 2003. [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0961

[6] "NetBSD repeated TIOSCTTY IOCTL buffer overflow vulnerability," CVE, Sep 2002. [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1490

[7] "Samba function reply_nttrans vulnerability," SecuriTeam, Jul 2003. [Online]. Available: http://www.securiteam.com/exploits/5TP0M2AAKS.html

[8] "Pine email header parsing vulnerability," SecuriTeam, Sep 2003. [Online]. Available: http://www.securiteam.com/exploits/5DP0D1PB5Y.html

[9] "GNU mailutils imap4d remote integer overflow vulnerability," SecurityFocus, Sep 2004. [Online]. Available: http://www.securityfocus.com/bid/11198/

[10] "Apache mod_auth_radius malformed RADIUS server reply vulnerability," SecurityFocus, Jan 2005. [Online]. Available: http://www.securityfocus.com/bid/12217/

[11] "Putty 0.53b SFTP client packet parsing integer overflow vulnerability," SecurityFocus, Feb 2005. [Online]. Available: http://www.securityfocus.com/bid/12601/

[12] "Samba directory access control list remote integer overflow vulnerability," SecurityFocus, Dec 2004. [Online]. Available: http://www.securityfocus.com/bid/11973

[13] "Mailutil-0.6 imap4d remote integer overflow vulnerability," SecurityFocus, May 2005. [Online]. Available: http://www.securityfocus.com/bid/13763/

[14] "libtiff STRIPOFFSETS integer overflow vulnerability," iDEFENSE lab, Dec 2004. [Online]. Available: http://www.idefense.com/intelligence/vulnerabilities/display.php?id=173

[15] "libtiff directory entry count integer overflow vulnerability," iDEFENSE lab, Dec 2004. [Online]. Available: http://www.idefense.com/intelligence/vulnerabilities/display.php?id=174

[16] ANSI, *ISO/IEC 9899: Programming Langauges - C*, 1999.

[17] *Ada95 Language Reference Manual*, ISO/IEC, 1995.

[18] B. Pierce, Ed., *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.

[19] B. C. Pierce, *Types and Programming Languages*. The MIT Press, 2002.

[20] "CVE (version 20040901)," CVE. [Online]. Available: http://www.cve.mitre.org/cgi-bin/cvekey.cgi?keyword=integer

[21] C. Cowan, C. Pu, D. Maier, and etc., "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks," in *USENIX Security Symposium*, 1998, p. 63C77.

[22] J. Condit, M. Harren, S. McPeak, and etc., "Ccured in the real world," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, June 2003.

[23] D. Brumley, D. Song, and J. Slember, "Towards automatically eliminating integer-based vulnerabilities," Carnegie Mellon University, Tech. Rep. CMU-CS-06-136, Mar 2006.

[24] G. Necula, S. McPeak, S. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of C," in *Proc. Conference on Compiler Construction*, 2002.

[25] ——, "CIL version 1.3.3," http://manju.cs.berkeley.edu/cil/, 2005.

[26] D. Dyer, "The top 10 ways to get screwed by the "c" programming language," http://www.andromeda.com/people/ddyer/topten.html, 2003.

[27] O. Horovitz, "Big loop integer protection," Phrack Inc., Dec 2002. [Online]. Available: http://www.phrack.org/phrack/60/p60-0x09.txt

[28] D. LeBlanc, "Integer handling with the C++ SafeInt class," Jan 2004. [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure01142004.asp

[29] M. Howard, "An overlooked construct and an integer overflow redux," Sep 2003. [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure09112003.asp

[30] M. Howard *et al.*, "Safe integer arithmetic in C," Feb 2006. [Online]. Available: http://blogs.msdn.com/michael_howard/archive/2006/02/02/523392.aspx

[31] P. H. I. Systems, "SPARK/ada," http://www.praxis-his.com/sparkada/sparkprojects.asp.

[32] J. Barnes, *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, 2003.

[33] G. C. Necula, S. McPeak, and W. Weimer, "CCured: type-safe retrofitting of legacy code," in *Proceedings of the Symposium on Principles of Programming Languages*, 2002.

[34] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *USENIX Annual Technical Conference*, 2002.

[35] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, ser. Lecture Notes in Computer Science, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[36] Y. Xie and A. Aiken, "Saturn: A SAT-based tool for bug detection," in *Proceedings of the 17th Conference on Computer Aided Verification*, 2005.

[37] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill, "CMC: A prgramatic approach to model checking real code," in *Proceedings of the 5th symposium on operating systems design and implementation*, 2002.

[38] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *Programming Language Design and Implementation (PLDI)*, 2001.

[39] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival, "The ASTREE static analyzer," http://www.astree.ens.fr/.

[40] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," in *Symposium on Operating System Principles*, 2001.

[41] S. Hallem, B. Chelf, Y. Xie, and D. Engler, "A system and language for building system-specific, static analyses," in *Proceedings of the* 24th *International Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[42] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[43] S. Muchnick, *Advanced Compiler Design and Implementation*. Academic Press, 1997.

[44] K. Redwine and N. Ramsey, "Widening integer arithmetic," in *Proceedings of the* 13th *International Conference on Compiler Construction (CC)*, 2004.

[45] C. Hawblitzel, E. Wei, H. Huang, E. Krupski, and L. Wittie, "Low-level linear memory management," in *Proceedings of the* 2nd *workshop on Semantics, Program Analysis and Computing Environments for Memory Management*, 2004.

[46] T. Freeman and F. Pfenning, "Refinement types for ML," 1991.

[47] S. Bhatkar, R. Sekar, and D. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Proceedings of the 14th USENIX Security Symposium*, August 2005.

[48] J. Yang, T. Kremenek, Y. Xie, and D. Engler, "MECA: an extensible, expressive system and language for statically checking security properties," in *Proceedings of the 10th ACM conference on Computer and communication security*. ACM Press, 2003, pp. 321–334.

A Full Subtyping Rules

Table 7 shows the full sub-typing rules we use in RICH. Each rule is read as an implication: when the preconditions on the top of the bar are satisfied, the formula on the bottom of the bar is true. A safe expression has a valid type, i.e., a type that can be derived via the rules. Here, Γ is the typing store that maps a variable name or expression to a type. The types in Γ are built via the declared C types. T-SUB introduces the sub-typing semantics, and says if our typing store Γ says variable t is of type σ, and σ is a subtype of τ, then t is also of type τ. We also add the standard reflexive (T-REFL) and transitive (T-TRANS) rules. T-FIELD just states that, when the programmer references a structure field, the expression uses the declared type for that field.

$$\frac{\Gamma \vdash t : \sigma \quad \sigma <: \tau}{\Gamma \vdash t : \tau}\ \text{T-SUB} \qquad \frac{}{\sigma <: \sigma}\ \text{T-REFL} \qquad \frac{\sigma <: \upsilon \quad \upsilon <: \tau}{\sigma <: \tau}\ \text{T-TRANS}$$

$$\frac{\Gamma \vdash s.i : \sigma \quad \sigma <: \tau}{\Gamma \vdash s.i : \tau}\ \text{T-FIELD} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \&t : \text{ref}\ \tau}\ \text{T-REF} \qquad \frac{\Gamma \vdash t : \text{ref}\ \tau}{\Gamma \vdash *t : \tau}\ \text{T-DEREF}$$

$$\frac{}{\text{unsigned} <: \text{uint8_t} <: \text{uint16_t} <: \text{uint32_t} <: \text{uint64_t}}\ \text{T-UNSIGNED}$$

$$\frac{}{\text{signed} <: \text{int8_t} <: \text{int16_t} <: \text{int32_t} <: \text{int64_t}}\ \text{T-SIGNED}$$

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash (\tau)e : \tau}\ (\text{T-UPCAST})$$

Table 7. Typing rules for statically-safe integer operations.

$$\frac{\Gamma \vdash e : \sigma \quad (\tau)e \quad \sigma <: \tau \quad e \rightsquigarrow e'}{(\tau)e \rightsquigarrow (\tau)e'}\ \text{R-SAFE} \qquad \frac{\Gamma \vdash e : \sigma \quad (\tau)e \quad \sigma \not<: \tau \quad \sigma, \tau <: \mathbb{Z} \quad e \rightsquigarrow e'}{(\tau)e \rightsquigarrow (\tau)\text{let}\ x : \sigma = e'\ \text{in}\ \text{CHECK}_{(\tau)\sigma}(x)}\ \text{R-UNSAFE}$$

$$\frac{\sigma \not<: \tau}{\text{CHECK}_{\tau,\sigma}(x) \equiv \text{if}\ \tau_{\text{min}} \leq x \leq \tau_{\text{max}}\ \text{then}\ x\ \text{else error}}\ \text{D-CHECK}$$

Table 8. R-UNSAFE rewrites unsafe casts by inserting dynamic checks: U-S-CHECK for unsigned to signed casts, S-U-CHECK for signed to unsigned casts, and D-CHECK for down-casts. R-SAFE is added for completeness: it leaves safe expressions as-is.