Towards Automatically Eliminating Integer-Based Vulnerabilities

David Brumley **Dawn Song** **Joseph Slember**

March 2006
Revision of original paper from December, 2005
CMU-CS-06-136

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Over 100 C integer vulnerabilities have been publicly identified to date, some of which have resulted in serious disasters such as rocket malfunction. C integer vulnerabilities can arise when one integer type is cast to another incompatible integer type. The rules which determine integer cast safety are cumbersome, lengthy, and sometimes unintuitive. As a result, it is common to find thousands of potentially unsafe casts in even moderately sized programs. Despite the importance of writing safe and secure programs, the burden of correctly using (often necessary) integer casts is placed squarely on developers.

We show that well-known sub-typing theory commonly found in type-safe languages can effectively an automatically be applied to protect against most integer casting vulnerabilities in C. We implement our techniques in a tool called PICK which statically detects potential integer vulnerabilities and inserts the necessary dynamic checks to prevent exploits. Our experiments (a) confirm potentially unsafe integer operations are rampant in source code, indicating the potential number of vulnerabilities is great, (b) show the introduced checks protect vulnerable programs, (c) show no manual modifications are needed in most cases, and (d) the inserted checks do not introduce measurable overhead. Thus, our approach and techniques provide a practical, efficient, and automatic method for protecting against integer vulnerabilities for even large programs written in C.

This work is supported by grants from the National Science Foundation.

Keywords: computer security, integer vulnerability, integer overflow, integer conversion error, software security

1 Introduction

The semantics of integer operations in C are complex and unintuitive to many, leading to insidious bugs and vulnerabilities due to ignored or misunderstood boundary and conversion conditions. An integer cast converts between different integer types, and when misused can cause serious vulnerabilities. Although there is a body of literature offering sage advice on how to program securely by avoiding pitfalls with integer operations, there has been very little being done to *automatically* secure existing C programs. The 179 known integer-based vulnerabilities [15] — most of which are integer casting bugs — serve as a testament to the clear need for techniques that defend against integer vulnerabilities. Integer casting bugs have even been responsible for huge disasters, such as the Ariane 5 rocket explosion which was caused by a conversion from a 64-bit floating point to a 16-bit signed integer [14]. The number of known vulnerabilities is likely the tip of the iceberg; our experiments indicate that potentially unsafe integer casts are rampant in programs. Of our tested programs (Section 4), the number of potentially unsafe casts range from 600 to almost 5000. Automatic techniques are clearly required to handle potentially unsafe casts found at this scale.

Motivating situation. A system administrator has downloaded an open-source application he would like to install. The administrator is unlikely to be familiar with the details in the code, but wants to protect his system from exploitation in case their are bugs in the code. For example, the system administrator can compile the code with stack-guard to protect from buffer overruns [4]. We wish to provide a similar tool for protecting against integer vulnerabilities. Our goal is to allow the administrator to protect his system from integer vulnerabilities in the application while requiring little if any code changes (the typical system administrator is likely not an expert programmer) nor sacrificing performance.

Possible approaches. One approach to fixing integer vulnerabilities is to raise a compile-time warning for each potential vulnerability and let the programmer fix each one. However, this approach seems impractical due to the sheer number of warnings. Another approach is to translate the C code into a type-safe variant, e.g., Cyclone [11] or CCured [18]. However, this option may not be practical in many settings, such as for performance-critical applications or when the user isn't intimately familiar with the code. Yet another approach is to try and weed out warnings for safe code. Our evidence suggests that the number of actual bugs is an order of magnitude less than the number of warnings. However, any tool that finds all bugs must be conservative, thus will generally have a high false positive rate in which a programmer will again be faced with a large number of warnings. Ultimately, manually fixing bugs is unavoidable. However, it would be useful in many situations to have an *efficient* and *automatic* approach for *protecting* against (not just detecting) any *potential* vulnerability.

Integer vulnerabilities. As mentioned, most integer vulnerabilities are due to unsafe casts. However, previous work does not adequately address protecting against unsafe casts. There are two casting categories: *sign conversions* where a signed integer can be converted to an unsigned integer (or vice-versa), and integer *precision conversions* where the number of bits used to represent the integer is changed. At a high level, the problem with sign conversions is the sign bit of a signed integer becomes the most significant bit in an unsigned integer (and vice-versa). As a result, negative signed integers become large unsigned integers (and vice-versa), leading to unintended program behavior. Precision conversions can cause a loss of precision via truncation when converting a value of a larger precision type to a smaller precision type, again leading to unintended behavior.

There are about a dozen rules in the ANSI C99 [1] standard determining the effects of a conversion via integer casting [1]. These rules define the semantics of a conversion based upon an integer ranking system. In

[1]Many of the C99 rules appear in paragraph form instead of as precise statements, so it is difficult to judge the exact number of rules.

many scenarios, these rules are complex and thus easily misunderstood. For example, it is easy to confuse whether $5U - 15$ is -10 or 4294967286 based upon the ranking rules (the answer is the latter for reasons detailed in Section 2.1). In other scenarios, a given conversion is defined as implementation-specific. For example, these rules define when an integer type is converted to another integer type where the value cannot be represented by the new type, the result is a signed integer that is implementation-defined. Unfortunately, this implementation-defined behavior can also lead to bugs and vulnerabilities.

This paper. We protect against integer casting vulnerabilities by rewriting unsafe casts as dynamic safety checks. As we will see, integer vulnerabilities are either overflow vulnerabilities or casting vulnerabilities, the former of which are already addressed by modern compilers (Section 3). At a high level, we address the larger problem of casting vulnerabilities by using sub-typing relationships to define integer cast safety. For example, up-casting an integer from a smaller-precision type to a larger-precision type is a safe sub-typing relationship since a larger precision integer can always represent the smaller precision integer. Down-casting from a larger to smaller precision violates the typing rules, and thus is not safe. However, down-casting (and other potentially unsafe type conversions) are rampant in source code, and therefore it would be naive to believe developers will manually address each potential unsafe integer cast. Therefore, we introduce formal rewrite rules which enable automatic source-to-source translation where unsafe casts are rewritten as safe dynamic checks. The dynamic checks raise an error only when a cast is unsafe at runtime.

Contributions. Our main contribution is we demonstrate automatic techniques for defending against a wide class of integer vulnerabilities in a formal framework. We show that by applying sub-typing theory we can detect *and protect* against a large class of integer vulnerabilities. We have implemented a tool called PICK to validate that our light-weight approach is practical and prevents integer vulnerabilities.

Specifically, we:

- Provide formal semantics for safe C integer casts. Our semantics replace the cumbersome and unintuitive C99 specifications with 2 simple sub-typing rules.
- Introduce rewrite rules that turn type unsafe (and semantically unsafe) casts into type-safe dynamic checks. The correct check to insert does not require expensive analysis, and thus scale to any size program.
- Implement a prototype called PICK (Preventive Integer Checks) to evaluate our approach and techniques.
- Demonstrate through experiments that potentially unsafe integer casts are rampant in source code, indicating the number of known vulnerabilities may be the tip of the iceberg.
- Show the introduced checks for unsafe casts protect vulnerable programs. The resulting program is semantically equivalent to the original program. Our experiments confirm our approach and techniques prevent real exploits against real vulnerabilities from working.
- Show our approach is fully automatic in most cases. 1 manual modification was needed out of thousands of automatically inserted checks. The 1 modification was needed because the programmer had inserted a similar check which handled the unsafe cast in an application-specific manner.
- Show the inserted checks do not introduce any measurable overhead, and are therefore practical to apply to production code.
- Additionally, our techniques uncover and protect against many portability bugs.

2 Integer Security Vulnerabilities

In this section we begin by providing a description of integer operations, focusing on the ANSI C99 specification. As we will see, the complexity of the C99 specification contrasts with the simplicity of our approach using sub-typing. We then outline integer vulnerabilities. Our work applies to both explicit casts and implicit casts (coercions) inserted by the compiler.

Notation: Instead of using basic C type names such as "unsigned" and "signed long long", we adopt the more descriptive C99 syntax for clarity, shown in Table 6 in Appendix A, throughout this paper.

2.1 Integer Representations and Conversion

The representation for all integers except for uint8_t is implementation specific. Note char is a type of integer, and can be signed or unsigned. Values of type uint8_t are represented with a single byte in binary notation. Most PC architectures use 2's complement to represent all other integer types. Different representations may cause portability bugs, e.g., in 1's complement representation there is both +0 and -0 which may not be correctly handled by the code.

An unsigned type uint$n$_t can represent any value between 0 and $2^n - 1$. The *precision* of an integer is the number of bits for representing a value excluding the sign bit (and any padding bits), and is simply n for an unsigned integer (page 39 [1]). The width of an integer is the precision plus any sign bits. A signed type int$n$_t can represent any value between -2^{n-1} and $2^{n-1} - 1$. The precision for signed integers is $n - 1$, while the width is n, e.g., the precision of a int8_t is 7 bits, although 8 bits are used to represent any value. Maximum and minimum values for all signed integers are defined in `limits.h`.

Often programmers will convert from one integer type to another via a *cast*. A compiler will also insert implicit casts (coercions) whenever the types in an expression or statement do not agree. Our techniques are applied after all casts have been inserted, including those automatically inserted by the compiler. According to C99, the semantics of a cast between two different integers relies on the *rank* of the integer. In particular, C99 defines about a dozen rules for determining the rank of an integer, a summary of which is (page 42 [1]):

- No two signed integer types shall have the same rank, even if they have the same representation.
- The rank of a signed integer type with greater precision is higher than signed integer types with less precision.
- The rank of an unsigned integer type is the same as the corresponding signed integer type.
- Ranking is transitive: if T1 has rank greater than T2 has rank greater than T3, then T1 has rank greater than T3.

A *precision conversion* cast may increase or decrease the precision of an integer. C99 defines an integer promotion, commonly called *up-casting*, as a cast from a lower precision type to a higher precision type (without changing the sign type). Similarly, we define an integer *down-cast* as a cast from a higher precision to a lower precision type. Demotion is defined in C99 as implementation-specific, and is usually carried out via truncation.

An integer *sign conversion* occurs when a signed integer type is cast to an unsigned type, or vice-versa. In each case the integer value bit pattern is preserved across casting. As a result, a negative integer type results in a very large unsigned integer, since the sign bit is set. Similarly, a large positive unsigned value may become negative. Although the bit pattern is preserved and no data is lost, sign conversions result in vulnerabilities when programmers do not anticipate these corner-case effects. For example, a programmer may cast a signed integer x to an unsigned integer y, and then later test if y is greater than some value. The programmer may not anticipate the case where $x < 0$ leads to a large y value.

Precision and sign conversions may be either explicit (via an explicit cast operation in the code) or implicit. The rules for conversion in C99 (page 45 [1]) are as follows:

1. If both operands have exactly the same type, no conversion is necessary.

2. If both operands are of the same integer kind (both signed or both unsigned), then the type with a smaller rank is promoted, i.e., up-casted.
3. If an operand with the unsigned type has rank greater (i.e., greater precision) than the signed type, the result is the type of the unsigned integer. Conversely, if the signed type has greater rank, the unsigned operand is converted to the type of the signed operand.
4. Otherwise, both operands are converted to the unsigned integer type.

C99 leaves many behaviors implementation-specific, such as down-casting. In general, C99 has this to say about conversions:

> *"When a value with integer type is converted to another integer type other than _Bool, if the value can be represented by the new type, it is unchanged.*
>
> *Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.*
>
> *Otherwise, the new type is signed and the value cannot be represented in it; either the result is implementation defined or an implementation-defined signal is raised."* [1]

For example, the expression `5U-15` is not -10, but 4294967286 because 15 is (implicitly) cast to an unsigned integer (rank rule 3 above), the result type will be unsigned, $-10 \bmod 2^{32} = 4294967286$ (paragraph 2 from C99 above).

2.2 Security Vulnerabilities with Integer Types

C integer vulnerabilities can be divided into two categories: integer wrapping vulnerabilities and integer casting vulnerabilities. Wrapping occurs when the result of an arithmetic operation produces a value that is greater (resp. less than) than can be stored in the fixed-width register. Wrapping vulnerabilities caused by arithmetic operations are already handled by popular compilers such as gcc (compiled with the optional -ftrapv flag) and Visual C++ (via the /RtCc flag). [2]

Therefore, we focus on the remaining previously unaddressed case of casting vulnerabilities, which fall into two categories: integer sign casts and integer precision casts [3]. Most known integer vulnerabilities are casting vulnerabilities, including the OpenSSH integer vulnerability [31] which has led to thousands of compromised machines. Indeed, many wrapping vulnerabilities are only symptomatic of an earlier unsafe down-cast which our approach would protect against.

Integer Sign Conversion Vulnerabilities. Integer sign conversions may result in vulnerabilities when (1) a negative signed integer is cast to unsigned, becoming a large value, or (2) a large positive unsigned integer is cast to a signed integer, becoming negative. Consider the following code (discovered by our analysis in bash-1.14.6), which ironically attempts to be a safe version of `malloc` by always checking whether memory allocation was successful:

```
char * xmalloc (int32_t bytes){
  char *temp = (char *)malloc (bytes);
  if (!temp)  memory_error_and_abort ();
  return (temp);  }
```

[2] We are unaware of a compile flag that will issue a warning for all types of integer casting bugs. Neither the default gcc's compile flags, nor -Wall or -pedantic, detect many simple casting bugs.

[3] Adding rewriting rules that check for overflow and underflow in a single coherent system is trivial using our approach and infrastructure. We do not duplicate previous work, thus do not discuss these checks.

The relevant detail is `malloc` takes a `uint32_t` argument, but is here provided with a signed `int32_t` argument. This particular case may lead to a denial of service because when called with a negative value a huge amount of memory is allocated. Another common example which often leads to a vulnerability is `memcpy`, whose prototype is:

```
void *memcpy(void *dest, const void *src, unsigned int len);
```

If a signed integer with a negative value is passed in as `len`, it will become a large positive number. This will lead to a buffer overflow when `dest` is not large enough to hold the converted `len` bytes of `src`. Notable examples of integer overflows involving `memcpy` include PuTTY [27] and Apache mod_auth_radius [26].

Integer Down-cast Vulnerabilities. An integer cast may increase (up-cast) or decrease (down-cast) the precision of the representation. Increasing the precision is always safe, and usually accomplished by zero-extending the casted value. However, decreasing the number of bits is potentially unsafe. An example of a typical down-casting vulnerability is:

```
1   uint16_t len = strlen(string):
2   char *buf = malloc(len);
3   strcpy(buf, string);
```

On line 1, `strlen` returns a 32-bit integer, which is down-cast to a 16-bit integer. As a result, a string of length 2^{16} will result in `len` = 0. The `strcpy` on line 3 can then be exploited with a standard stack-smashing attack. Again, such vulnerabilities often appear when trying to secure software, such as in the OpenSSH CRC32 vulnerability where a down-casting error leads to a vulnerability, ironically in code meant to detect certain types of cryptographic attacks [31].

3 Our Approach: Strong Integer Typing

We define integer casting in terms of sub-typing rules, where safe casts are well-typed and unsafe casts are not well-typed. Intuitively, sub-typing allows us to succinctly express when one integer type can safety be cast as another integer type. We use 2 sub-typing rules to express the dozen or so C99 rules. Unsafe integer expressions are statically rewritten (via formal rewriting rules) as well-typed dynamic safety checks. Each dynamic check makes sure the cast is value-preserving, i.e., the value of the variable before the cast is the same as the value after the cast. We check all casts: both those implicitly inserted by the compiler (i.e., coercions) and explicitly provided.

This section introduces the formalism needed in order to rigorously define when and which safety checks to insert, as well as the safety they afford. We begin by introducing our typing rules, and discuss our types for basic integer operations. We then introduce our dynamic checks for potentially unsafe casts. Then, we discuss more complex types such as structures and pointers.

3.1 C Integer Sub-typing Rules for Safe Integer Casts

Table 3.1 contains our typing rules for safe integer operations. 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. An unsafe integer expression has an invalid type.

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

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

$$\frac{}{\text{unsigned} <: \text{uint8\_t} <: \text{uint16\_t} <: \text{uint32\_t} <: \text{uint64\_t}}\ \text{T-UNSIGNED}$$

$$\frac{}{\text{signed} <: \text{int8\_t} <: \text{int16\_t} <: \text{int32\_t} <: \text{int64\_t}}\ \text{T-SIGNED}$$

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

Table 1: Our typing rules for safe C integer operations.

3.1.1 Basic Sub-typing Relationships: T-SUB, T-REFL, T-TRANS

The intuition in our setting behind a sub-typing relationship, written $\sigma <: \tau$, is any value described by type σ is also described by type τ. In our formulation, we have sub-types such as `uint8_t` $<:$ `uint16_t` because $\{0...2^8 - 1\} \subset \{0...2^{16} - 1\}$. In general, smaller precision integers are sub-types of larger precision integers since a larger precision can express any value of a smaller precision.

T-SUB in Table 3.1 introduces the sub-typing relationship to C. Here, Γ is the typing store that maps a variable name or expression to a type. [4] The rule T-SUB is the basic sub-typing rule, 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.

3.1.2 Sub-typing Rules for Safe Casts: T-UNSIGNED, T-SIGNED, T-UPCAST

Our approach defines two basic types: *unsigned* and *signed*. Different precisions within a type become sub-types. We express casts in terms of sub-typing where smaller precisions are sub-types of larger precisions. T-UNSIGNED and T-SIGNED in Table 3.1 express the base sub-typing relationship for integers, while T-UPCAST states that we can up-cast (ascribe) to an expression e of type σ a super-type τ. For example:

```
// Cast explicit or implicitly inserted by the compiler
uint8_t b; uint16_t a = (uint16_t) b;
```

is safe because it is well-typed:

$$\cfrac{\Gamma \vdash \texttt{b} : \text{uint8\_t} \qquad \cfrac{}{\text{uint8\_t} <: \text{uint16\_t}}\ \text{T-UNSIGNED}}{\Gamma \vdash (\text{uint16\_t})\texttt{b} : \text{uint16\_t}}\ \text{T-UPCAST}$$

Note T-SIGNED and T-UNSIGNED, along with T-UPCAST eloquently replace the dozen or so rules for determining the rank and result of rank conversion that appear in C99. We believe this simplicity makes our approach appealing. Also note that T-TRANS can be applied for two or more up-casts, e.g., `uint8_t` being up cast to `uint32_t`.

3.2 C Integer Rewriting Rule for Unsafe Casts

Down-casts and sign conversions are not within the type system, and therefore potentially unsafe. We rewrite potentially unsafe casts as runtime safety checks on the operands. The resulting expression with the safety

[4] The types in Γ are built via the declared C types.

$$\frac{\Gamma \vdash e : \sigma \quad (\tau)e \quad \sigma \not<: \tau \quad e \rightsquigarrow e'}{(\tau)e \rightsquigarrow (\tau)\text{let } x : \sigma = e' \text{ in } \textsc{check}_{(\tau)\sigma}(x)} \text{ R-UNSAFE}$$

$$\frac{\Gamma \vdash e : \sigma \quad (\tau)e \quad \sigma <: \tau \quad e \rightsquigarrow e'}{(\tau)e \rightsquigarrow (\tau)e'} \text{ R-SAFE}$$

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

$$\frac{\sigma \not<: \tau \quad \text{signed} <: \sigma \quad \text{unsigned} <: \tau}{\textsc{check}_{(\tau)\sigma}(x) \equiv \text{if } 0 \leq x \leq \tau_{\text{max}} \text{ then x else error}} \text{ S-U-CHECK}$$

$$\frac{\sigma \not<: \tau \quad \text{unsigned} <: \sigma \quad \text{signed} <: \tau}{\textsc{check}_{(\tau)\sigma}(x) \equiv \text{if } x \leq \tau_{\text{max}} \text{ then x else error}} \text{ U-S-CHECK}$$

Table 2: 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.

check is well-typed. [5] The particular check depends upon whether the unsafe cast is a sign conversion cast or a down-cast. Table 2 gives our rewriting and safety check rules.

3.2.1 General Rewriting Rule for All Unsafe Casts: R-UNSAFE

We introduce rewrite rules for potentially unsafe casts of expressions. Suppose in $(\tau)e$, e is of the type σ which is not a subtype of the cast type τ, i.e., $\sigma \not<: \tau$. For example, when assigning an unsigned to signed integer, the signed integer is σ and is cast to type unsigned integer τ. We translate e to an expression that performs the proper safety check during evaluation.

R-UNSAFE in Table 2 states that an unsafe cast $(\tau)e : \sigma$ where e evaluates to some other expression e' is rewritten statically to another cast where e is evaluated to a value x, which is checked via CHECK$_{\tau,\sigma}$. CHECK$_{\tau,\sigma}$ is a function which returns the value x or calls an `error()` function. The check functions (S-U-CHECK, U-S-CHECK, and D-CHECK) are different for each type of unsafe cast: unsigned to signed conversions, signed to unsigned conversions, and down-casts. R-SAFE is included for completeness: for safe casts, no rewrite is necessary.

3.2.2 Specific CHECK$_{\tau,\sigma}$: D-CHECK, U-S-CHECK, and S-U-CHECK

Down-casts. A down-cast from an expression e of type σ of higher precision to τ of lower precision (e.g., uint32_t to uint16_t) requires a check if the value of e when evaluated is preserved with the smaller precision type τ, i.e., the value of e "fits" inside the type τ. A down-cast is essentially a sub-typing relationship backward: $\sigma \not<: \tau$ but $\tau <: \sigma$.

CHECK$_{(\tau)\sigma}$ for down-casting is given as D-CHECK in Table 2. The rule states that an error is raised if the value x of e is larger than the maximum value τ_{max} or smaller than the minimum value τ_{min} of integer type

[5] For brevity, we omit several uninteresting rules that are technically needed to show this.

τ. The pre-condition $\sigma \not<: \tau$ and $\tau <: \sigma$ are needed to ensure we only apply this rule when the precision is changed, but not the sign (sign changes are handled by S-U-CHECK and U-S-CHECK).

For example, consider the code:

```
uint32_t b;
uint16_t a = (uint16_t) b;
```

Here σ = `uint32_t` and τ = `uint16_t`. Since `uint32_t` $\not<:$ `uint16_t`, the rewriting rule R-UNSAFE applies:

$$\frac{\Gamma \vdash b : \text{uint32\_t} \quad \text{uint32\_t} \not<: \text{uint16\_t} \quad b \rightsquigarrow b'}{(\text{uint16\_t})b \rightsquigarrow (\text{uint16\_t}) \text{ let } x : \text{uint32\_t} = b \text{ in } \text{CHECK}_{(\tau)\sigma}(\text{x})} \text{ R-UNSAFE}$$

Further, since the sub-typing is backward, we use the D-CHECK rule:

$$\frac{\text{uint32\_t} \not<: \text{uint16\_t} \quad \overline{\text{uint16\_t} <: \text{uint32\_t}}\ \text{T-UNSIGNED}}{\text{CHECK}_{(\tau)\sigma}(x) \equiv \text{if } 0 \leq x < 2^{16} - 1 \text{ then x else error}} \text{ D-CHECK}$$

The rewriting of the example given the formal rules is then:

```
uint32_t b, c;
uint32_t x = b;
if (0 <= x && x <= 2^16 − 1) x; else error();
uint16_t a = (uint16_t) b;
```

In our implementation, we output the equivalent:

```
uint32_t b;
if (b > 2^16 − 1) error();
uint16_t a = (uint16_t) b;
```

Sign conversion casts. The sign bit must be checked for conversions between signed and unsigned integers. We divide *check*$_{(\tau)\sigma}$ for sign conversions into two cases as shown in Table 2: S-U-CHECK where a signed integer is cast to an unsigned integer, and U-S-CHECK where an unsigned integer is cast to a signed integer.

U-S-CHECK is similar to D-CHECK with the exception that τ is signed and σ is unsigned, while in D-CHECK both are either signed or unsigned. Although the resulting check is the same, we find it useful to logically separate out unsigned to signed conversions from down-casts. The signed to unsigned conversion check S-U-CHECK need only check that the sign bit is not set, i.e., $x \geq 0$.

For example, the S-U-CHECK and U-S-CHECK will rewrite the following:

```
int32_t i32; uint32_t u32;
i32 =  u32;
u32 =  i32;
```

as:

```
int32_t i32; uint32_t u32;
if(u32 > 2^31 − 1) error(); // U−S−CHECK
i32 = (int32_t) u32;
if(i32 < 0) error();   // S−U−CHECK
u32 = (uint32_t) i32;
```

A formal derivation showing this rewriting for each cast is similar to that given for down-casts above, where the main difference is the preconditions for S-U-CHECK and U-S-CHECK are satisfied instead of D-CHECK.

3.3 Dynamic Safety Error Detection: `error()`

Our translation results in an `error()` when a cast for a particular value will be unsafe. Runtime checks have a long history; for example, in Java the sub-typing rule for arrays of subclasses is unsafe, which is handled by introducing dynamic safety checks [21]. In Java, run-time safety violations cause an exception, which results in termination unless caught. Another example is divide-by-zero errors in C++, which cause uncaught runtime exceptions in most programs.

At a high level, when an unknown error is encountered there are two choices: attempt to correct the error or abort execution. The user can define `error()` to implement either of these choices. Of course manually fixing the bug is the best choice, but not an option in many situations. Others have explored aborting the current function [29] or returning a random result [22] when an error is encountered, which allows the program to continue executing. However, both these approach are not fail-safe, and thus not useful in many security-conscious scenarios.

Since integer vulnerabilities often lead to privilege escalation, e.g., an integer vulnerability due to casting in OpenSSH leads to remote root access [31], we believe the safest action is for `error()` to abort the program. Although aborting may lead to denial of service attacks, it does prevent more serious problems such as privilege escalation, arbitrary code execution, etc., and is the approach taken by similar safety tools, e.g., stack-guard [4]. Therefore, we currently abort the program when a safety violation is detected. We could easily change this to print out a warning, or throw an exception (via a signal and signal-handler). Warnings are unsafe because they do not prevent the error. Exceptions may be interesting in some scenarios since it could be used to trigger additional analysis or to "hack" around known conversion problems.

3.4 Complex Types

Structures. T-FIELD in Table 3.1 handles integral fields within structures in the obvious way: if the type of field i is σ, and $\sigma <: \tau$, then via sub-typing, i is also of type τ.

For example, in

```
struct { uint32_t u32; uint16_t u16; } foo;
foo.u32 = (uint32_t) foo.u16;
foo.u16 = (uint16_t) foo.u32;
```

In the first assignment, `foo.u16` is of type `uint16_t`, thus the assignment is a safe up-cast. However, in the second assignment `uint32_t` $\not<:$ `uint16_t`, and a down-cast check must be inserted.

Note unions can be handled in a manner similar to T-FIELD: each union field member is declared with a type. The sub-typing relationships then range over that declared type.

References and Dereferences. For each integral type τ, Ref τ denotes the type of a pointer to type τ. Integral reference and dereferences are handled via generic typing rules T-REF and T-DEREF, respectively.

T-DEREF in Table 3.1 states that if we have a pointer to an integer type, then a dereference yields an object of the pointed-to type, e.g., if `p` is of type `uint16_t *`, then `*p` is of type `uint16_t`. Therefore:

```
uint8_t v; uint16_t *p;
v = *p;
```

is not safe, and rewritten as:

```
if(*p > 2^8-1) error();
v = *p;
```

Pointers. The above rules check that pointer reads and writes are correct with respect to the declared type, i.e., if the programmer writes with one type and reads with another compatible type, we assume it is intentional. For example, we assume:

```
uint32_t *u32; uint16_t *u16;
....
*u32 = *u16;
```

is correct since `uint16_t` is a sub-type of `uint32_t`.

The above assumption does not necessarily hold, i.e., the programmer could simply have mixed up their types. Safe pointer assignment *could* be handled by adding the standard type safety rule:

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\text{Ref } \sigma <: \text{Ref } \tau} \ \text{T-RefSub}^*$$

This rule states that a reference of type σ is a subtype of a reference to type τ if τ and σ are sub-types of one another, i.e., in our semantics $\tau = \sigma$.

However, we find the standard type safety rule too restrictive compared to the benefit of strict integer type safety for typical C programs. For example, T-RefSub* would disallow the following typical code:

```
1  uint16_t *u16 = malloc(10);
2  ...
3  uint32_t *u32 = u16 + 9;
4  uint8_t val = *u16;
```

Clearly line 4 is potentially unsafe since the 8-bit `val` may be too small for the 16-bit `*u16`. Our semantics without T-RefSub* will insert a proper check on this line. However, line 3 is also unsafe. For example, a subsequent write through `u32` is 4-bytes long due to its type, while `u16+9` only has 2-bytes available. If we want complete safety and accept T-RefSub*, then line 3 is also unsafe.

We have found that even without implementing T-RefSub* all integer vulnerabilities we know of are protected. An informal analysis of known integer vulnerabilities indicates they arise primarily when integers are used as indexes or to determine the size of allocated memory, both of which are checked with our rules during dereference. The overall intuition is integer vulnerabilities arise because the integer value is not what was expected in a localized computation.

Since T-RefSub* is overly restrictive and breaks many legitimate programs, we do not currently implement it. For example, the above rule would break typical networking code found in many of our examples:

```
1  uint16_t *buf; uint16_t u16; int8_t i8;
2  ....
3  *buf = i8; // a int8_t = 1 byte is written
4  ...
5  u16 = *buf; // but uint16_t = 2 bytes are read
```

Here, `buf` is intended to be an uninterpreted 2-byte buffer, where reads are writes are of the correct though mis-matched packet field type.

It appears very difficult to ensure type safety in this code without tagging each memory write with the corresponding type, and checking each subsequent read. This tagging would likely incur a huge overhead with what appears little additional value. Even if this overhead was acceptable, it would likely be impossible to derive a generic rule that works for all programs. For instance, in the above example on line 5 the programmer may have wanted to read 2 `uint8_t`'s concatenated together on line 5, or may have wanted to read 1 `uint8_t` and cast it to a `uint16_t`.

We therefore do not use T-REFSUB* by default, but leave it as an optional extension. We leave as an area for further research a light-weight check that will prevent integer vulnerabilities that arise through pointer casts. We remark that any such research would also have to handle casts from `void *` to be complete since `malloc`, `read`, `write`, etc. all return `void *` which are then cast to the "right" type.

3.5 Where Checks Are Inserted

A pre-processing step identifies all expressions in which a cast is needed. This step is already performed by the compiler: the type of each expression is needed to generate the proper code. If the cast is not explicitly provided by the programmer, an implicit cast is inserted (i.e., coerced) by the compiler. We then perform a typing derivation to determine which casts are safe, and which are unsafe. Unsafe casts are rewritten via the R-UNSAFE rule.

Note that function call sites act as an assignment from actuals to formals, thus may also need a check. For example, in:

```
1  void f(uint8_t v) { v++; }
2  void foo(){
3    uint16_t u16;  int8_t i8; int16_t i16; uint32_t u32;
4    f(u16);
5    i8 = ( (u16 * u32) + i16 );   }
```

We insert checks on line 5 to make sure the cast of `u16` from a `uint16_t` to a `uint8_t` is safe. We will also check the sub-expression on line 5. Note that on line 5 the right-hand side has a mix of unsigned and signed integers. `foo` is rewritten as:

```
...
 if(u16 > 2^8 - 1) error();
 f(u16);
 if( ((u16 * u32) + i16) > 2^7 - 1) error();
 i8 = (int8_t) (( (uint32_t) u16 * u32) + (uint32_t)i16);
```

4 Implementation and Evaluation

4.1 Implementation

We have implemented a tool called PICK (Preventive Integer Checks) which automatically inserts the necessary checks to prevent integer casting vulnerabilities. PICK is implemented using CIL [16, 17], a C analysis and source-to-source translation framework written in OCaml. CIL takes as input the source code to a program, performs several semantic-preserving simplifications, and then produces a typed intermediate representation (IR). Our analysis is performed 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.

We use the type symbol table provided by CIL to decide when to insert the appropriate checks as given by the rules in Section 3. Note our analysis is at the expression level, and therefore does not require intra or inter-procedural analysis, i.e., we do not need to merge all files together. As a result, the overhead for analyzing the code and introducing the proper checks is negligible. The steps taken by PICK are:

Name	Vuln	# S-U-CHECK	# U-S-CHECK	# D-CHECK	# Full	# Total
Apache 2.0.35	Y	2058	1120	368	368	3914
Bash 1.14.6	Y	160	370	92	5	627
Coreutils 5.0	Y	461	252	113	34	860
MLTerm 2.9.1	Y	853	1143	417	442	2855
OpenSSH 2.2.0p2	Y	320	281	33	43	677
Gzip 1.2.4	N	203	270	36	97	606
OpenSSL 0.9.7	N	2647	1124	826	373	4997

Table 3: S-U-CHECK (U-S-CHECK) is the number of signed to unsigned (unsigned to signed) checks inserted. D-CHECK is the number of down-cast checks inserted. Full is the number of places both a sign check and down-cast check are inserted. This experiment shows potential integer vulnerabilities are rampant in code.

1. Pre-process the input, e.g., inline `#include` statements, expand macros, etc. CIL performs this step automatically by invoking the C pre-processor.
2. Create a typing store which maps all expressions and statements to a type. Insert implicit casts when necessary, e.g., in an expression of mixed types, a statement where the type of the lhs is different from that of the rhs, etc. CIL performs this step automatically.
3. For each cast, both explicit or implicit, perform the following steps:
   (a) In cast $(\tau)e$, τ is the cast type and let σ be the type of the expression e.
      - If $\sigma \not<: \tau$, create a check c. Insert the check just prior to the evaluation of e (e.g., on the line before e).
      - If $\sigma <: \tau$, do nothing.
      - If $e \rightsquigarrow e'$ where e' is some sub-expression, then recurse and apply typing rules on e'.
4. Insert the `error()` function, which in our implementation calls `exit(-42)`.

We perform all our tests under Linux, using gcc 3.4, though CIL and our transformations work also under other operating systems and compilers such as MSVC under Windows. Our CIL module is approximately 500 lines of OCaml.

4.2 Evaluation

We perform two quantitative experiments: measuring the number of checks that need to be inserted in various programs, and measuring the overhead of the inserted checks. We also provide qualitative evidence of the type of common unsafe and unportable integer expressions within code.

4.2.1 Number of checks inserted

We ran PICK on a number of programs to measure how frequently potential integer vulnerabilities occur. Table 3 lists our results. The table includes the program name, whether there is a previously known vulnerability or not, and the number of down-casts and sign conversion checks that were inserted. We include non-vulnerable programs into the analysis to get a sense of how many checks need be inserted for many different kinds of software. The S-U-CHECK column indicates the number of signed-unsigned checks, U-S-CHECK the number of unsigned-signed checks, and D-CHECK the number of down-casting checks.

The last column is the total number of checks. The "Full" column indicates the number of full range checks where both a down-cast and a sign conversion take place in the same instruction.

A "Full" check occurs when the typing rules are recursively applied resulting in both a sign check and a down-cast check. As a result, a full check may require two comparison with both an upper and lower bound of the resulting integer type. For example:

```
uint8_t u8; int16_t i16;
u8 = i16;
```

A full check is needed here because `i16` may be signed *or* too large for type `uint8_t`. The full check is `i16` > 0 `&&` `i16` $< 2^8 - 1$.

Our experiments indicate that potential integer vulnerabilities and bugs are rampant in source code. These numbers also support the idea that issuing compile-time warnings is not practical due to the sheer number of casts, supporting our run-time check approach.

4.2.2 Error Analysis

For vulnerable programs, we confirmed that the checks prevented all vulnerable programs from being exploitable, i.e., zero false negatives. This is expected since our approach results in type-safe integer operations. Most vulnerabilities seem to be due to down-casts, while most portability bugs seem due to signedness conversions. Our analysis uncovered 11 additional portability bugs in OpenSSH, and 1 in gzip. One common problem we found is programmers expect type char to be analogous to a byte. However, C99 specifies only *unsigned* char's are analogous to a byte (Section 2.1). This particular problem is cited by others such as [7, 6, 25] , often as a member of the top 20 in C bugs. In each case we modified the source code to remove otherwise implicit and compiler-specific casts [6].

In all our experiments, we only found 1 example of a false positive – a check inserted that fails at runtime but was not needed – in bash 1.14.6. The relevant code is:

```
1  void remove_trailing_whitespace( char *userstr) {
2    int i = strlen(userstr) - 1;   // strlen returns an unsigned integer
3    while(i > 0 && whitespace(string[i])
4      i--; ...
```

The salient detail is `strlen` returns an unsigned integer, causing the arithmetic operation of line 3 to be `signed = unsigned - signed`. C99 dictates the result is therefore an unsigned integer. Therefore, when `userstr` is empty, `strcpy` returns 0 and the expression results in $0 - 1 = 4294967295$, which when cast to a signed integer results in -1. This is exactly the sort of casting that causes vulnerabilities. Therefore, at runtime we raise an error exception.

The code, in a convoluted manner, performs a similar check. As mentioned, the rhs expression on line 3 will be 4294967295 unsigned, which equals -1 signed. On line 4, the condition $i > 0$ guards against executing the loop on such a casting error. Therefore, the net effect is the check is not needed and subsequently the run-time exception is an error. We believe ugly code such as this should be rewritten anyway because the reliance on corner cases results in difficult to maintain code.

It is easy to imagine many cases where our approach may insert checks that are already handled more gracefully by the existing code. Our experiments suggest these are rare in real code, however. Therefore, annotating such casts with an attribute that indicates the check should not be performed seems the best solution.

[6] This problem is so wide-spread that gcc supported a -funsigned-char flag that will make all char's unsigned by default. Interestingly, gcc does not seem to support a flag that reports these portability errors.

	GCC	PICK	
No Opt.	15.473s	15.556s (+0.536%)	15.474s (+0.006%)
-O4 Opt.	2.967s	5.586s (+88.27%)	2.971s (+0.134%)

Table 4: Micro-benchmark measuring the running time (averaged over 5 runs) in seconds for a tight loop executing a cast from `uint32_t` to `int16_t`.

Additional modifications. Recall that C99 states that during a cast $(\tau)e$ where τ is an unsigned integer, the value of e will be repeatedly added or subtracted until it is within the precision of τ (Section 2.1). In other words, $(\tau)e = e \bmod 2^t$ where t is the precision of τ, i.e., a truncation. However, a similar cast when τ is of a signed type is implementation defined.

The current version of PICK follows the type-checking rules and ignores this nuance of the C99 standard: unsigned and signed truncation are both treated in exactly the same. The reason we made this decision is truncation errors are an artifact of the programmer not protecting against corner cases, and this also seems like a corner case likely to be abused. Although changing this behavior is trivial, we believe relying on implicit truncation is a dangerous practice at best. We found 2 and 6 places in gzip and OpenSSH respectively that relied on this nuance of the standard and had to be manually changed. For example, when OpenSSH reads in a packet it processes it byte by byte can be modified:

```
uint8_t byte; uint16_t val;
byte = val; // original code relying on truncation of unsigned bytes
byte = val & 0xff;  // modified code explicitly specifies the truncation
```

We stress the changes are required because of an implementation decision that could easily be changed if warranted.

4.2.3 Performance

Micro-benchmarks. In order to gain a better understanding of the cost of PICK 'ing code, we ran several micro-benchmarks and tests. First, we manually inspected the assembly code generated by PICK . An example of a check and the corresponding x86 disassembly is given in Appendix B Figure 1. Each check is only a few extra instructions, which are unlikely to make a measurable difference given the speed of modern processors.

Second, we created a small program which executes a tight loop casting a `uint32_t` to a `int16_t` (therefore performing both a U-S-CHECK and a D-CHECK) about 2 billion times. The results are shown in Table 4. The table shows three scenarios: a base case where the code is compiled directly with gcc, the case where the code is compiled with PICK , and finally the case where the code is ran through CIL without inserting checks and compiled with gcc. For each scenario, we tested compiling the code with no optimizations and full optimizations (gcc -O4), and report the average over 5 runs. We found that simply running the code through CIL changed the performance slightly, indicating CIL's default simplifying (though semantic-preserving) transformations can alter the run-time characteristics of code. Running the code through PICK without optimization incurred a negligible overhead: only .5%.

When compiled with full optimizations, PICK 'ed code incurred about an 88% overhead. The PICK checks constitute about 1/3 of the code statements. It appears that most of the degradation is not due to the extra instructions: they can be pipelined with the rest of the loop body, but that the introduced checks interfere with loop optimizations. Our micro-benchmark stresses this corner case. Regular programs rarely exhibit such tight loops consisting entire of cast operations, thus are unlikely to have a similar performance degradation.

Name	Vuln	Size Diff	Performance
gzip-1.2.4	N	+4.6%	0.44% faster
openssh-2.2.0p1	Y	+0.9%	7.08% faster

Table 5: There is no measurable overhead performance for the dynamic safety checks.

Macro-benchmarks. We measured the performance of PICK 'ed code to determine (a) the run-time overhead of the inserted checks and (b) the increase in the size of the binary on real code. Table 5 shows our results. The reported numbers are averages over 5 independent runs.

The performance checks do not appreciably increase the size of the compiled code. In fact, table 5 shows that PICK 'ed code actually ran faster than the original source code. We initially found this very peculiar: why should code with the extra check instructions run faster? We double checked our experiments several times, each with about the same result. In the end, we find there are two contributing factors. First, modern processors do not simply run code sequentially instruction by instruction. Instead, they have an entire optimization engine that dynamically optimizes running code through pipelining, thread speculation, cache layout optimizations, etc. For example, adding a few additional `no-ops` can drastically change the timing characteristics of code [3] – both for the better and for the worse. Second, CIL performs various simplifying transformations that may affect performance, i.e., code that has been run through CIL without any transformations then compiled with gcc may run faster or slower than code compiled directly with gcc.

Overall, PICK 'ing code should not affect performance significantly. Each check is only a few instructions, can easily be pipelined, and uses operands already locally referenced. Therefore, except in extreme cases, PICK should only have negligible impact on performance.

5 Related Work

Our techniques are drawn directly from type theory. A good introduction to type theory is provided in [21], which discusses several of the issues in type conversions such as down-casting. Using dynamic checks when static type safety cannot be discerned also appears in other languages such as in Java [21]. However, we are the first to show these techniques are practical for securing existing C programs.

Cyclone [11] and CCured [18] are type-safe versions of C. Our work differs from such approaches by only considering a subset of C – namely integer operations – and rewriting the program in terms of C instead of a safe alternative. Since Cyclone and CCured are much more ambitious, many different problematic C constructs may have to be manually translated. Our limited application is more appropriate for securing existing programs against only integer vulnerabilities, e.g., possibly in combination with other approaches for combating buffer overflows, format string errors, etc.

Our analysis creates checks for all potentially unsafe integer operations. However, in security one may only be concerned with unsafe operations under the attackers control. Taint analysis [10, 28, 12] could be used to isolate only those potentially unsafe operations along "tainted" program paths that may be under an attackers control.

There are several static checkers such as Splint [9] and meta-compilation [8] which can be used to find integer casting bugs by writing appropriate rules. We could use these tools to locate potential integer vulnerabilities. However, these tools do not introduce checks that will *protect* against exploits of the vulnerability.

Many C integer vulnerabilities are subsequently exploited via buffer overflow attacks, on which there is a wide-body of research. Static checks for detecting such attacks include [4, 13, 24]. Stack smashing attacks can also be detected dynamically, such as with [2, 5, 19, 20, 23, 30]. Interestingly, it appears detecting integer exploits themselves dynamically is troublesome due to the lack of type information at the x86 level, i.e., it may be impossible to tell whether a cast is occurring. Overflow attacks, on the other hand, can be detected dynamically. However, not all integer overflows are necessarily malicious. For example, the x86 instruction set actually has a `jo` instruction for jumping on overflow which some compilers take advantage of in legitimate code transformations and layouts.

6 Conclusion

We have presented an approach using well-known type theory for eliminating integer casting vulnerabilities. Our experiments re-confirm that potentially unsafe integer operations are a real problem. Our implementation and evaluation shows that integer casting vulnerabilities can be fixed by automatically inserting light-weight checks in the code, and that the resulting fixes do not effect performance. Our techniques protect against previously unaddressed integer vulnerabilities. The widespread application of our techniques are realistic in production compilers such as gcc, and would result in eliminating a large class of integer vulnerabilities.

References

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

[2] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *USENIX Annual Technical Conference 2000*, 2000.

[3] David Brumley and Dan Boneh. Remote timing attacks are practical. In *Proceedings of the 12th USENIX Security Symposium*, August 2003.

[4] Crispin Cowan, Calton Pu, Dave Maier, Jonathon Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[5] Jedidiah R. Crandall and Fred Chong. Minos: Architectural support for software security through control data integrity. In *To appear in International Symposium on Microarchitecture*, December 2004.

[6] Peter Darnell and Philip Margolis. *C: A Software Engineering Approach*. Springer, 2005.

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

[8] Dawson Engler, David Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating System Principles*, 2001.

[9] David Evans and David Larochelle. *Splint Manual*, 2003.

[10] Jeffrey Foster, Manuel Fahndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1999.

[11] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *USENIX Annual Technical Conference*, 2002.

[12] Robert Johnson and David Wagner. Finding user/kernel pointer bugs with type inference. In *Proceedings of the* 13^{th} *USENIX Security Symposium*, 2004.

[13] Richard Jones and Paul Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the Third International Workshop on Automated Debugging*, 1995.

[14] J. L. LIONS. Ariane 5: Flight 501 failure. Technical report, Report by the Inquiry Board, 1996.

[15] MITRE. Common vulnerability and exposures (CVE) database. http://www.cve.mitre.org.

[16] George Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C. In *Proc. Conference on Compiler Construction*, 2002.

[17] George Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL version 1.3.3. http://manju.cs.berkeley.edu/cil/, 2005.

[18] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the Symposium on Principles of Programming Languages*, 2002.

[19] Nicholas Nethercote and Jeremy Fitzhardinge. Bounds-checking entire programs without recompiling. In *Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, Venice, Italy, January 2004. (Proceedings not formally published.).

[20] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.

[21] Benjamin C Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[22] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel Roy, Tudor Leu, and William Beebee Jr. Enhancing server availability and security through failure-oblivious computing. In *Operating System Design & Implementation (OSDI)*, 2004.

[23] Tim J Robbins. libformat. http://www.securityfocus.com/tools/1818, 2001.

[24] Olatunji Ruwase and Monica Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February 2004.

[25] Robert Seacord. *Secure Coding in C and C++*. Addison-Wesley, 2005.

[26] Securiteam. Apache mod_auth_radius remote integer overflow. http://www.securiteam.com/unixfocus/5BP0B1PELW.html, 2005.

[27] Securiteam. Multiple integer overflow vulnerabilities in putty sftp. http://www.securiteam.com/windowsntfocus/5TP0Q0KEUI.html, 2005.

[28] Umesh Shankar, Kunal Talwar, Jeffrey Foster, and David Wagner. Detecting format-string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

Our Syntax (Unsigned)	Equiv. Base Type	Our Syntax (Signed)	Equiv. Base Type
uint8_t	unsigned char	int8_t	signed char, char
uint16_t	unsigned short	int16_t	signed short, short
uint32_t	unsigned, unsigned int, unsigned long	int32_t	int, signed int, singed long, long
uint64_t	unsigned long long	int64_t	signed long long, long long

Table 6: The C99 notation, which we use for clarity. The corresponding base C type is also given.

[29] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference*, 2005.

[30] G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of ASPLOS*, 2004.

[31] Michael Zalewski. Ssh1 crc-32 compensation attack detector vulnerability. `http://www.coresecurity.com/common/showdoc.php?idx=81&idxseccion=10`, 2001.

A Types

Table 6 shows a correspondence from integer types commonly found in C and the integral types used throughout this paper and in C99 [1].

B Disassembly of Inserted Check

Figure 1 shows the disassembly of one of the inserted checks. Note that it is only a few instructions long, references only variables in the expression anyway, thus can be easily pipelined.

```
/* int32_t a; */
/* uint8_t b; */
/* if(a < 0 || a > USHRT_MAX) */
8048384: cmpl   $0x0,0xfffffffc(%ebp)
8048388:  js     8048395 <main+0x2d>
804838a: cmpl   $0xffff,0xfffffffc(%ebp)
8048391: jg     8048395 <main+0x2d>
8048393: jmp    804839f <main+0x37>
/*  exit(1); */
8048395:  sub    $0xc,%esp
8048398:  push   $0x1
804839a:  call   80482b0 <exit@plt>
/* b = a; */
804839f:  mov    0xfffffffc(%ebp),%eax
80483a2:  mov    %al,0xfffffffb(%ebp)
```

Figure 1: Disassembly of a sign and width check. Instructions 804839f and 80483a2 perform the assignment. The check is 5 instructions (8048384-8048393).