Automated Code Repair Based on Inferred Specifications

William Klieber, Will Snavely
Software Engineering Institute
Carnegie Mellon University
{weklieber,wsnavely}@cert.org

***Abstract*—Techniques for automated code repair have the potential for greatly aiding in the development of secure and correct code. There are currently a few major difficulties confronting the development and deployment of tools for automated repair; we examine these and briefly explore possible solutions. To give a flavor of what automated repair might look like, we discuss in detail three types of proposed automated repair: (1) repairing inequality comparisons involving integer overflow to behave the same as if unlimited-bitwidth integers were used, (2) inserting memory bounds checks where needed, using dynamic analysis to infer tightest correct bounds, (3) inserting missing authorization checks in a client-server application based on an inferred access control policy.**

I. Introduction

Static analysis has been increasingly used in real-world settings to find potential bugs. However, static analyzers typically issue many false alarms, and it is a burden to thoroughly review each finding. In recent years, researchers have been developing techniques for not only detecting certain classes of bugs but also automatically repairing them. We claim that automated repair is the way forward for developing code free of common security weaknesses. In particular, we claim that (1) many security bugs follow common patterns, (2) by recognizing such a pattern, it is often possible to make a reasonable guess of the developer's intention, i.e., the desired specification, and (3) it is often possible to automatically repair the code to satisfy this inferred specification.

To give an example, a C programmer who writes "`malloc(n * sizeof(t))`" likely intended for `malloc` to try to allocate enough memory to hold `n` objects of type `t`. This code can be repaired to check for integer overflow, and if it occurs, to behave the same as if `malloc` returns `NULL`.

Sometimes it might not be possible to unambiguously infer the desired behavior. For example, an analysis might find that a program might write past the end of a buffer. In this case, an automated repair might be to insert a bounds check and abort execution if it fails. This isn't a full repair (from a correctness standpoint), but it does fix the security vulnerability.

In fact, many types of automated repairs consist of inserting a check for an error condition. A difficulty in such cases is how to handle the error when it occurs. The simplest thing to do is just call `abort()`, but such a repair is likely not to be accepted by the developer as is. A more desirable course of action would be to try to identify other error-handling code that can be re-used or adapted. For example, if a function f fails to check whether a pointer returned from `malloc` is `NULL` before dereferencing it, an automated repair tool might look to see if any 'nearby' code (e.g., a callee of f, or a sole caller of f) handles a `malloc` failure, and if so, to re-use this error-handling code. Research in 'self-healing' software has also explored the problem of how to handle errors unanticipated by the developer [27], [26].

Some approaches to automated repair have used genetic algorithms, notably GenProg [15], which uses dynamic analysis on a user-provided test suite consisting of known-good test cases and failing test cases. PACHIKA [5] likewise uses a dynamic analysis on a test suite with good and bad cases. Such techniques have shown promising experimental results, but the accuracy is not guaranteed enough for a developer to blindly accept such repairs without manually verifying. Recently an automated repair for memory leaks has been developed based on sound static analysis, providing a guarantee that the repair doesn't break any good traces [9]. Jin et al. developed an automated repair, based mostly on static analysis, that targets atomicity violations in concurrent programs [13]. A recent survey of work in automated repair may be found in [17].

In section II, we discuss general difficulties faced by automated repair of source code. In the remainder of the paper, we discuss three types of often-encountered bugs and propose techniques for automated repair of them. These bugs concern (1) integer overflow, (2) memory bounds, and (3) access control. We have a preliminary implementation of some integer overflow repairs, though they have not been experimentally evaluated. The proposals for memory bounds and access control do not (yet) have corresponding implementations.

II. Benefits and Difficulties in Repair of Source Code vs Object Code

From a practical standpoint, perhaps the greatest difficulty in developing an automated repair specifically at the source-code level (as opposed to a source-to-binary compiler pass) is *the IR↔AST mapping problem*. Most static analyses work best and/or are far easier to write on an intermediate representation (IR) such as LLVM's IR, and often it is advantageous to perform certain optimization passes such as converting to static single assignment (SSA) form. However, the actual source-to-source repair necessarily must be done at the level of the abstract syntax tree (AST) or similar high-level representation that has a direct correspondence to the original source code.

The trouble then is that, once we have found a spot in the IR that needs to be repaired, existing tools provide no way to map it back to the AST. Furthermore, it is not just a matter of engineering effort: some useful optimization passes destroy the ability to provide an unambiguous mapping from the IR back to the AST. Further research and tool development in this area would be beneficial to automated repair.

The use of macros in C provides an additional obstacle on top of the IR↔AST mapping program. The newest development version of the ROSE compiler framework [24] has some support to modifying source code while preserving macros. Templates in C++ are another, perhaps more vexing, source of difficulty.

The need to avoid breaking good traces of program is yet another difficulty facing automated repair. If there is even a 1% chance that a 'repair' will break a program in some hard-to-detect way, developers will be hesitant to apply it without manually investigating its correctness. For this reason, automated repair operating on object code must be extremely conservative, whereas if a source-to-source tool is only 95% confident of a repair, it can tentatively make the repair and flag it for human review.

Despite all the troubles associated with source-to-source repair, it provides significant advantages over repairing the IR as a compile-time pass. Automated repair often involves guesswork, so the ability of developers to audit or improve the repairs is often essential. Another point is that static analysis is often fickle; if repair is done at compile time, minor modifications to program source code can unpredictably change what the tranformation does. In certain highly-regulated industries, use of automatic repair on object code is not legally certified for safety-critical systems, but a tool can still suggest changes (at the source-code level) for an engineer to independently evaluate.

III. Integer Overflow

In this section, we describe an approach for automating detecting and repairing integer overflow[1] in C that leads to memory violations. In particular, we consider two situations where integer overflow can lead to a memory violation:

1) Memory allocation, such as `malloc(`n`)`, where the calculation of n can overflow, resulting in a too-small allocation and subsequent out-of-bounds memory access.
2) Integer overflow involving indices or bounds of an array. (We only consider arithmetic where both operands are of integer type; pointer arithmetic is outside the scope of this work.)

Let us use the term "memory-related integer overflow" to describe an integer overflow in either of the two above cases. Subsection III-A describes in detail how we detect memory-related integer overflow.

[1]We use the term "integer overflow" to describe an arithmetic operation (addition, subtraction, multiplication, or division) using n-bit modular arithmetic that produces a different answer than would be produced using normal (non-modular) arithmetic. Some authors use a narrower definition and distinguish between *overflow* and *underflow*; we do not make this distinction.

Our key assumption (inferred specification) is that memory-related integer overflow is undesired and that comparisons potentially involving memory-related integer overflow (e.g., "`offset + count < buffer_size`") should instead behave the same as if unlimited-bitwidth integers were used. For memory allocation (which internally compares the amount of memory requested to the amount of address space unallocated), this means that, if the program requests an amount too big to be represented as a `size_t`, the allocation should fail (as opposed to allocating the requested amount modulo 2^n, where n is the width of `size_t` in bits).

If an overflowed value is used to index into an array (or equivalently, if it is added to a pointer which is then dereferenced), we are unable to fully infer the desired behavior. For software that isn't safety-critical, our proposed repair tool can be instructed to simply check for overflow and call `abort()` if it is detected. Alternatively, it can insert a check for overflow and leave it up to the user to write the error-handling code.

Integer overflows that are unrelated to memory are not necessarily undesired. For example, many cryptographic and hashing functions are designed to employ modular arithmetic. Consequently, we ignore integer overflow that is unrelated to memory access/allocation.

We have several strategies for repairing overflows that reach comparison operators:

1) Promote the operands to a higher bitwidth. Some compilers support a `__int128_t` type. This is the preferred strategy if it fully repairs the overflow and meets the portability requirements of the codebase.
2) For unsigned addition and subtraction, a standard idiom can be used to check for overflow. For example, given unsigned integers `x` and `y`, the expression `(x + y < x)` is true iff `x + y` overflows. Then "`x + y < foo`" can be repaired as "`x + y < foo && x + y < x`". This solution is used if portability is required with compilers that lack a 128-bit integer type.
3) For unsigned integers, if the result of an overflowing operation is only used in subsequent addition or multiplication (not subtraction or division), then the quantity is monotonically non-decreasing in subsequent arithmetic (except for multiplication by zero). Thus, unlimited-bitwidth arithmetic can be emulated by using *saturation arithmetic*, i.e., replacing an overflowed value with the greatest representable value (`SIZE_MAX` for type `size_t`). If the declared types of variables are smaller than `size_t`, they are changed to `size_t`.[2]

As an example of the saturation-arithmetic strategy, the code in Figure 2 is repaired in Figure 3 using the convenience function `umul` defined in Figure 1. One complication is comparison of an overflowed value to the greatest representable value (`SIZE_MAX`). For example, in Figure 2, if the value of the function parameter `max_img_size` is `SIZE_MAX` and the computation of `size` overflows, then the expression

[2]We ignore the complication that may arise if the value is later compared to a value of type `__int128_t` or if `size_t` is smaller than `uintmax_t`.

```
static inline size_t umul(size_t x, size_t y) {
    size_t ret;  bool flag;
    flag = __builtin_mul_overflow(x, y, &ret);
    if (flag) {ret = SIZE_MAX;}
    return ret;
}
```

Fig. 1. Convenience function `umul` for saturating multiplication. The function `__builtin_mul_overflow(x,y,p)` does the multiplication `x*y` and stores the result in `p`. It returns `true` if overflow occurred, and `false` otherwise. This builtin function is available in `gcc` and `clang`.

```
size = hdr.width * hdr.height * hdr.bytes_per_pixel;
if (max_img_size != 0 && size > max_img_size) {
    return ERR_TOO_BIG;
}
bitmap = malloc(size);
```

Fig. 2. Code with possible integer overflow.

```
size = umul(umul(hdr.width, hdr.height),
            hdr.bytes_per_pixel);
if (max_img_size != 0 && size > max_img_size) {
    return ERR_TOO_BIG;
}
bitmap = malloc(size);
```

Fig. 3. Repaired to emulate unlimited-bitwidth arithmetic. The only change is to replace occurrences of the "`*`" operator with function calls to `umul`. Note that `malloc(size)` internally compares the value of `size` to the largest available contiguous chunk of virtual memory, so if the value of `size` is `SIZE_MAX`, then `malloc` will behave as desired (return `NULL` and set `errno` to `ENOMEM`).

`size > max_img_size` should evaluate to true according to the semantics of normal (non-modular) arithmetic, but it doesn't in our repaired code. However, since this situation would require a more complex and less readable patch for dubious benefit, we do not do so.

Of the above strategies, all but the first (promoting the operands to a higher bitwidth) damage the readability of the code. Better compiler support for dealing with overflows would be needed to avoid this.

A. Detection of Undesired Overflows

We want to transform arithmetic with potential overflow only if it involves a variable that is associated with memory access or memory allocation; we say that these variables are *sensitive*. To determine which variables are sensitive, we use a flow-insensitive backward *taint analysis* on single static assignment (SSA) form:

1) Mark a variable as sensitive if it is added to or subtracted from a pointer or used as an array index.
2) Mark a variable as sensitive if it is compared with a sensitive variable for equality ("$=$"), inequality ("$<$", "$\leq$", "$\geq$", "$>$"), or disequality ("$\neq$").
3) For a statement of the form "v_1 := $v_2 \star v_3$", where "$\star$" denotes an arithmetic operation ("+", "−", "*", "/"), if any of v_1, v_2, or v_3 are sensitive, then mark all of them as sensitive.
4) For an assignment of the form "v_1 := v_2" or a phi node of the form "$v_1 = \Phi(v_2, ..., v_n)$", if any of the v_i variables are sensitive, then mark them all as sensitive.
5) Mark a variable as sensitive if it is used as an argument to a function and the corresponding formal parameter in the function definition is sensitive. For a system library function (e.g., `malloc`, `mmap`), consider the corresponding formal parameter to be sensitive if it has been manually designated as such.

The above analysis is similar to type inference. However, note that sensitivity is not propagated via memory loads and stores; e.g., the code "`*p = x; y = *p; z = arr[y];`" does not cause `x` to be marked as sensitive. To handle such cases, we require an optimization pass to transform the assignment "`y = *p`" to "`y = x`" before conducting the taint analysis. For this purpose, we can use the 'merging memory loads' transformation described in Section 5.4.2 of [32].

The modulo operator ("`%`") is intentionally excluded from propagating sensitivity, because in many common situations (e.g., implementation of a hash table), code that uses a value of the form `n%m` to index into an array is safe and correct even if the computation of `n` involved an arithmetic overflow.

To detect whether an arithmetic operation can potentially overflow, we use KINT [32], a static analyzer. KINT creates an abstraction of a function body using a simple memory model and unrolling each loop once. Idioms for checking for unsigned overflow (e.g., `x + y < x`) are recognized and replaced with intrinsic functions. For each arithmetic expression in the abstraction, KINT asks whether an integer overflow can happen. This question is formulated as an SMT problem and solved using Boolector [1].

B. Implementation

We have a preliminary implementation of an integer overflow repair tool that replaces potentially overflowing operations with saturating arithmetic.

This tool executes in three phases: (1) A build phase, (2) an analysis phase, and (3) a repair phase. The build phase involves compiling the desired source files through a proxy. This proxy collects information about each source file into a shared database, then routes the source file to an actual compiler (e.g., GCC). During this phase, a call graph is constructed, recording function definitions and their callees. The proxy is implemented with the ROSE compiler infrastructure.

The analysis phase leverages the information gathered from the previous phase to perform whole program analysis. For example, a set of sensitive function sinks is computed during this phase, using the call graph and a simple fixed-point algorithm. We begin with a set of known sensitive sinks, including `malloc`. We then search the call graph for functions that invoke a sensitive-sink function, and whose formal parameters influence the call. These functions are added to the set of sensitive sinks. This process is repeated iteratively until no

change is observed in the set. KINT is also invoked at this stage to identify potentially arithmetic overflows. The results of this analysis are recorded into the same database as before.

The repair phase uses the results of the whole program analysis to make modifications to the source files. For each file, sensitive sinks are enumerated. Then, for each sink, a backwards data flow analysis is used to find reaching arithmetic operations. If any of these operations were identified as potentially overflowing, then they are replaced with saturating alternatives. Arithmetic operations tainted by potentially overflowed values are also replaced. The repair phase is implemented with the ROSE compiler infrastructure.

We are currently in the process of evaluating this tool on real-world code bases, such as OpenSSL.

C. Related Work – Integer Overflow

IntPatch [33] and the AIR Integer Model [6] use source-to-binary transformations as part of a compiler to fix integer errors that could potentially lead to security vulnerabilities. Coker and Munawar [3] introduce source-to-source transformations to address programs with integer errors. However, their technique is not intended to be used fully automatically, as it does not try to distinguish desired overflow from undesired. Additionally, its sole repair for integer overflow is to trap into a user-defined error-handler, instead of repairing the program to emulate unlimited-bitwidth arithmetic. Logozzo and Ball [16] introduce an automated repair for integer overflow in intermediate calculations, such as rewriting `(a + b)/2` as `a + (b - a)/2` in cases where `a` and `b` are of signed integer type and are known to have non-negative values.

IV. MEMORY BOUNDS

In this section, we discuss an approach to automatically infer and insert bounds checks in source code. Bounds checks can help prevent invalid writes (which can corrupt memory) and invalid reads (which can leak sensitive information, as in the OpenSSL HeartBleed vulnerability (CVE-2014-0160)).

In C, it is sometimes difficult to determine what the proper memory bounds for a pointer should be. For a pointer in a region of memory returned by `malloc` (or by a custom memory allocator), the bounds of the memory region are also bounds on the pointer, but there might be better (tighter) bounds. For example, a pointer to an element of an array might be legitimately used to access the whole array, or might be properly constrained to just the single element.

Similarly, in some cases, a pointer to a sub-object inside a larger object should be constrained to the sub-object, as in Figure 4. The best upper bound of `acct->id` is `acct->id + sizeof(char[8])`, not `acct->id + sizeof(bank_acct)`. In other cases, a pointer to a sub-object is legimately used to access the containing object. A prime example of this occurs in the Linux kernel, where a linked-list header can occur inside a larger struct, as illustrated in Figure 5 and Figure 6. Linux has a macro `list_entry` for taking a pointer to this sub-object and returning a pointer to the containing object.

```
struct bank_acct {char id[8]; int balance;};
...
bank_acct* acct = malloc(sizeof(struct bank_acct));
strcpy(acct->id, "overflow...");
```

Fig. 4. Sub-object overflow (example adapted from [21])

```
struct list_head {
  struct list_head *next;
  struct list_head *prev;
};
```

```
struct task_struct {
  long state;
  pid_t pid;
  list_head sibling;
};
```

Fig. 5. Linked-list struct in Linux kernel

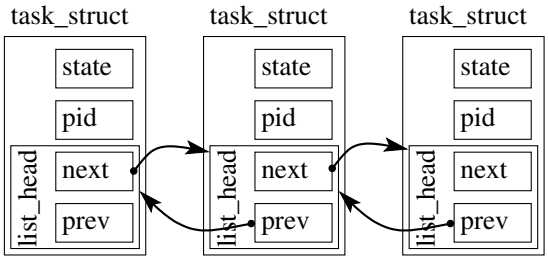

Fig. 6. Illustration of linked-list sub-object in Linux kernel

For a reusable buffer only partially filled with valid data (with the remainder of the buffer containing uninitialized or stale data), the bounds for reading should not exceed the valid portion, to avoid leakage of potentially sensitive information. This problem can even happen in memory-safe languages such as Java. For example, the Jetty web server (written in Java) had a vulnerability that could leak passwords, authentication tokens, and any other data contained in an HTTP request (CVE-2015-2080).

We will use static analysis to propose candidate bounds. Strategies to propose candidate bounds include:

1) Bounds of the region allocated by `malloc` (and custom allocators).
2) Bounds computed from use of the pointer. E.g., analysis of memory accesses within loops and limits of the loop.
3) Inference of function pre- and post-conditions. E.g.: If there are many callsites of a function `foo(int n, char *p, ...)`, and in most of these callsites, the bounds on `p` is the closed interval $[\mathtt{p}, \mathtt{p}+\mathtt{n}-1]$, then propose that in the other callsites, the same bounds should apply. The statistical method described in Section 9 of [10] can be used to estimate confidence in the inferred specifications and reject those for which the confidence is too low.
4) Invariants for structs. E.g., suppose that we discover that, in most of the program, one field of a struct supplies the bounds of another field of the struct. Then we guess that this is an invariant and violations of it are errors.
5) For reading from a re-usable buffer, propose that the

upper bound for reading is the most recently written position of the buffer. (This addresses the HeartBleed and Jetty vulnerabilities described above.) For greater confidence in such a bound, we will use taint analysis to determine if this bound also corresponds to a change of the source of the data.

After using static analysis to propose candidate bounds, we will use dynamic analysis to weed out too-strict candidate bounds by seeing what memory is actually accessed on good runs of the program:

1) We will instrument the program to record which, if any, candidate bounds checks fail.
2) We will run the instrumented program to collect 'good' traces, i.e., traces on which the program exhibits desired behavior. We will try to get as close to complete coverage of the code as possible. To do this, we can use test cases and/or run the program in a realistic test environment. The instrumented program will write to a log file to indicate which checks are violated, as well as statistical data on checks that succeed.
3) Based on the log data from the above step, candidate bounds will be divided into 3 categories:
   - Strongly supported: Many traces where the bounds check succeeded, with values distributed near the bounds, and no failed checks.
   - Likely incorrect: Some traces where the bounds check failed, and no evidence that these are bad traces.
   - Indeterminate: Insufficient log data about the check.

For candidates bounds in the "indeterminate" category, we will try to construct 'malicious' inputs that would violate the inferred bounds, and ask the developer to confirm or reject the candidate bounds.

If multiple candidate bounds are proposed for the same pointer, all will be evaluated by the dynamic analysis.

Finally, we will do the repair: In source code locations where the program does not already adequately check the bounds of memory accesses, we will repair the program by inserting code that checks the bounds and aborts if the check fails. (Of course, where possible, we will avoid inserting a bounds check inside an inner loop, and instead insert a single check above the loop that aborts if and only if any iteration of the loop would access memory out of bounds.)

This transformation can also be used to convert an existing C codebase to *Checked C*, an extension of C being developed by David Tarditi at Microsoft Research [31]. Checked C adds syntax for indicating the bounds of pointers, enabling the compiler to prevent out-of-bounds memory accesses. Unlike earlier work such as Cyclone [12] and CCured [22], Checked C is a strict superset of C (i.e., every valid C program is a valid Checked C program), allowing incremental adoption. Working in Checked C rather than ISO Standard C provides many of the benefits of a safer language such as Java while retaining the familiarity and other benefits of C. (However, Checked C does not fully guarantee memory safety: errors such as use-after-free and double-free are not covered.) In addition, static analysis can often determine that bounds checks are always satisfied, allowing dynamic checks to be omitted and guaranteeing correctness (not just absence of security vulnerabilities), which is important to safety- and mission-critical software.

A. Related Work – Memory Bounds

Rugina and Rinard [23] developed a static analysis to infer memory bounds. This analysis determines the bounds on what the program is able to access. It can detect potential accesses beyond the end of a `malloc`'d block, but it doesn't consider the complication that there might be better (tighter) bounds.

Dhurjati and Adve [7] developed a source-to-binary compiler pass that inserts bounds checks using fat pointers. It is not binary-compatible with existing third-party libraries.

SoftBound [21] also uses a compiler pass to insert bounds checks. It stores the bounds information in a separate region of memory, allowing it to interface with existing libraries. Typically SoftBound has a slowdown of 50%–60% [20].

Shaw, Doggett, and Hafiz [25] developed a source-to-source intra-procedural transformation for adding bounds checks for character arrays. This approach replaces raw `char*` pointers with fat pointers (using a struct called "`stralloc`"). However, this repair is only intra-procedural; only unaliased local variables (not global variables or pointers stored in heap-allocated memory) can be converted to fat pointers, and furthermore fat pointers are not passed across function boundaries.

Other work in this area includes [11], [28], [18], and [8].

V. Access Control

Let us consider a database-backed application that runs on a central server and talks to remote clients (such as web browsers, smartphone apps, etc.). Such an application has authorization logic (typically written in a web application language such as Ruby or PHP) that controls which users have access to which items in the database. Ordinary testing of the system may fail to reveal gaps in this server-side access-control logic. A gap would be defined as a failure to prevent a user from accessing data they are prohibited from accessing. That is, gaps represent differences between the application's enforcement and the security policy. An attacker may be able to exploit such gaps by creating a malicious client or even simply by entering hand-crafted URLs into the web browser. Our goal is to infer the intended access-control policy of the server application and automatically repair deviations from it. The vulnerability type addressed here appears on the OWASP list of the top ten web vulnerabilities (2013-A7 "Missing Function-Level Access Control") and falls under CWE-285.

Our fundamental assumption is that the desired access-control policy is expressed in the normal interaction between the client and the server. In other words, we assume that the desired access-control policy is to allow a given user to read (or respectively write) a data item if and only if the user can read (or resp. write) the data item using normal interactions of the client.

Some requests from the client to the server may include *authenticating* information such as a password or a login token, which we will assume that an attacker cannot guess.

As a motivating example, let us consider a collaborative document editing/viewing system, implemented as a web application backed by an SQL database. Each document has one or more authors, who are allowed to edit the document. In addition, each document is associated with zero or more teams, and each user is on zero or more teams. A user on a team associated with a document has permission to view the document (but not edit the document, unless the user is also an author of the document). The database for this application has the following schema:

1) Table `Document` has columns $\langle$`doc_id`, `doc_title`$\rangle$.
2) Table `DocAuthor` has columns $\langle$`doc_id`, `user_id`$\rangle$.
3) Table `DocTeam` has columns $\langle$`doc_id`, `team_id`$\rangle$.
4) Table `UserTeam` has columns $\langle$`user_id`, `team_id`$\rangle$.
5) Table `User` has columns $\langle$`user_id`, `password`$\rangle$.

We will use first-order logic to represent the database. For example, we will write "$\texttt{Document}(x, y)$" to denote an atomic proposition that evaluates to `true` iff there exists a row in the `Document` table whose `doc_id` field is x and whose `doc_title` field is y. As another example, the formula

$$\exists team\_id.\ \texttt{UserTeam}(user\_id, team\_id) \wedge \texttt{DocTeam}(doc\_id, team\_id)$$

is true iff there exists a $team\_id$ such that the `UserTeam` table has a row $\langle user\_id, team\_id\rangle$ and the `DocTeam` table has a row $\langle doc\_id, team\_id\rangle$. In terms of the business logic, this means that user $user\_id$ is on a team that has permission to view document doc_id.

We will write "$\texttt{ClientKnows}(table, column, key)$" to denote an atomic proposition that is true iff the client knows the secret corresponding to specified column of the row in $table$ whose primary key is key.[3] For example, $\texttt{ClientKnows}(\text{“}\texttt{User}\text{”}, \text{“}\texttt{password}\text{”}, 42)$ means that the client knows the password for the user whose `user_id` is 42.

We use an approach based on *model checking* [2] and *abstract interpretation* [4]. We create an abstraction of the client-server system. An abstract *state* of the client-server system will consist of the following information:

1) Server session variables (e.g., `$_SESSION` in PHP).
2) Cookies (stored client-side)
3) Set of requests available in the client user interface.

Formally: $State = Session \times Cookie \times AvailReqs$

Given a state s, we write "$s.Session$" to denote the $Session$ component of s, and likewise for $Cookie$ and $AvailReqs$. It may be noted that $State$ doesn't have components for either the database or client authentication knowledge, even though these are part of a concrete state of the system. This is because our analysis is parameterized by these quantities; we will symbolically analyze the system for all possible configurations of the database and client knowledge, and the three components of $State$ will be expressed in terms of them.

The $Session$ component of the state is represented as a formula in first-order logic. Static analysis is used to determine the set of session variables that are used by the application.[4] This set is exactly the set of free variables allowed in $Session$. For example, the following value for $Session$ indicates that the session variable `user_id` can be any user ID for which the client knows the password, and that $\texttt{IsAdmin} = 0$, and leaves all other session variables unconstrained:

$$\texttt{ClientKnows}(\text{“}\texttt{User}\text{”}, \text{“}\texttt{password}\text{”}, user\_id) \wedge IsAdmin{=}0$$

The special value `undef` is used to indicate that a variable is not assigned any value. For example, an empty session (no variables assigned a value) would be represented by the formula $\bigwedge_{v \in V}(v = \texttt{undef})$ where V is the set of all session variables. The $Cookie$ component is represented in the same manner as the $Session$ component.

The $AvailReqs$ component is likewise represented as a formula in first-order logic. Static analysis is used to determine the set of parameter names that are used by the application. HTTP POST requests have parameters that appear in the body of the request in addition to parameters in the URL. We create a free variable for each URL parameter; the name of the variable is the name of the URL parameter prefixed with "`url.`". For POST parameters, we use the prefix "`post.`". Finally, the $AvailReqs$ formula contains a free variable $BaseURL$ that contains the path portion of the URL. For example, the URL "`/edit-doc?doc_id=42`" would be represented by:

$$BaseURL = \text{“}\texttt{/edit-doc}\text{”} \wedge url.doc\_id = 42 \wedge \bigwedge_{v \in (ReqVars \setminus \{url.doc\_id\})} \Big(v = \texttt{undef}\Big)$$

where $ReqVars$ is the set of all request parameters used by the application (as determined by static analysis). Note that the formula explicitly requires that all parameters other than doc_id be absent. Given a request r, we say "r is in $AvailReqs$" iff r *satisfies* the formula $AvailReqs$ (makes the formula evaluate to `true`).

A *transition* from one state to another state encompasses the client sending a request to the server and the server sending back a response to the client. We define a transition relation T such that $T(s, r, s')$ holds true if the system can make a transition from state s to state s' via request r. Earlier, we spoke of "normal interaction" between the client and the server. We now define this more precisely: A transition from state s via a request r is considered a *normal interaction* iff the request r is in $s.AvailReqs$.

Given a request r and state s, let $db\_access(r, s)$ denote the *sensitive* database accesses performed during the server-side processing of r during a transition from s. An access is

[3] It may be noted that there is a potential namespace clash if a database table is named "ClientKnows". This can be resolved by, e.g., prefixing the name of every database table with "`DB.`".

[4] Variables whose names are created dynamically are not handled. If such variables are used and are relevant to access control, they represent a source of unsoundness for the proposed algorithm.

sensitive iff either it reads a field and sends the value to the client or it modifies the database. (A read isn't sensitive if the value doesn't get sent to the client.)

Static analysis is used to compute (an approximation of) the transition relation T and db_access. Both the server-side code and the client-side code (e.g., JavaScript for a web app) would need to be analyzed in general.

We use a formula in first-order logic to represent a set of sensitive database operations. The formula has (at most) the following free variables:

- op: the operation: `select`, `update`, `insert`, or `delete`.
- $table$: the name of the table being read or written.
- col: the column of the table being read or written.
- $pri_1, ..., pri_n$: the components of the primary key for the row of the database being read or written (ignored for `insert` and `delete` operations). Table `Document`'s primary key has a single component, `doc_id`. Table `DocTeam`'s primary key has two, `doc_id` and `team_id`.

For example, the SQL query "`select doc_id, doc_title from Document where doc_id=42`" corresponds to

$$op{=}\text{“}\texttt{select}\text{”} \wedge table{=}\text{“}\texttt{Document}\text{”} \wedge (col{=}\text{“}\texttt{doc\_title}\text{”} \vee col{=}\text{“}\texttt{doc\_id}\text{”}) \wedge pri_1{=}42$$

As another example, a database operation that allows the user to update the title of all documents for which s/he is an author:

$$op = \text{“}\texttt{update}\text{”} \wedge table = \text{“}\texttt{Document}\text{”} \wedge col = \text{“}\texttt{doc\_title}\text{”} \wedge \exists user\_id.\ \exists doc\_id.\ pri_1 = doc\_id \wedge \texttt{DocAuthor}(user\_id, doc\_id) \wedge \texttt{ClientKnows}(\text{“}\texttt{User}\text{”}, \text{“}\texttt{password}\text{”}, user\_id)$$

We write $Reach^k_{\text{UI}}(s)$ to denote the set of all states reachable from s in at most k transitions (limited to transitions available in the client UI):

$$Reach^1_{\text{UI}}(s) = \left\{ s' \mid \exists r.\ T(s, r, s') \wedge r \in s.AvailReqs \right\}$$
$$Reach^k_{\text{UI}}(s) = \bigcup_{s' \in Reach^{k-1}_{\text{UI}}(s)} Reach^1_{\text{UI}}(s') \qquad \text{for } k > 1$$

Let s_0 denote the starting state of the system. Consider a given transition bound k (say $k = 10$). We infer the intended access control policy by taking the disjunction of the allowed database operations for all requests available in the client UI of all states reachable within the transition bound:

$$IntendedAccess = \bigvee_{\substack{s \in Reach^k_{\text{UI}}(s_0) \\ r \in s.AvailReqs}} db\_access(r, s)$$

We compute an attacker's access similarly, except without the restriction that the request be available in the client UI:[5]

$$AttackerAccess = \exists r. \bigvee_{s \in Reach^k_{\text{UI}}(s_0)} db\_access(r, s)$$

[5]This can be improved by defining and using $Reach^k_{\text{atk}}$ for reachability via any transition, instead of $Reach^k_{\text{UI}}$, which is limited to transitions in the UI. Also, the attacker should be allowed to choose arbitrary values for $Cookie$.

We then ask if the attacker access exceeds the intended access:

$$\exists \{op, table, col, \overrightarrow{pri}\}.\ AttackerAccess \wedge \neg IntendedAccess$$

This can be answered by a first-order logic (FOL) solver such as VAMPIRE [14]. If the answer is yes, then the solver returns a request r that satisfies the formula. We then determine the condition under which r is available in the client UI, and we repair the codebase to add this condition as an authorization for processing r.

One source of difficulty is database denormalization, where the same underlying ground fact has multiple redundant representations. If such ground facts are relevant to authorization checks, then denormalization may cause false positives as well as slowing down the analysis.

Related Work. RoleCast statically identifies potentially sensitive operations such as database WRITEs that are unguarded by an authorization check [30]. In an experiment of 11 real-world PHP and JSP web apps, RoleCast found that 5 of the 11 apps had missing authorization checks that allowed attackers to write to database fields. This work has been extended to also suggest repairs [29]. However, this work is coarse-grained: it cannot handle situations where access to different fields in the database require different authorization checks. It also doesn't detect leakage of sensitive data to unauthorized users. Unauthorized disclosure of sensitive information is also a problem; [19] cites 9 such CVEs from 2010 to 2014 of well-known software packages and presents a dynamic analysis and web proxy to mitigate this problem.

VI. CONCLUSION

Automated code repair has the potential to significantly improve our ability to develop and maintain secure code by removing the burdensome manual effort that has traditionally been needed. A good problem suitable for automated repair is one with a common pattern that allow us to be highly confident in inferring what the specification of the program should be. Techniques for inferring a specification vary widely, including static analysis, dynamic analysis, and heavy-weight formal methods. Figure 7 summarizes inferred specifications and repairs discussed in this paper. Future reseach directions include tackling the IR↔AST problem.

	Inferred Spec	**Repair**
Integer overflow	Arithmetic for array bounds or indices should not overflow	Emulate unlimited bitwidth arithmetic where possible
Memory bounds	Infer desired bounds of memory	Insert missing bounds check, call abort() if the check fails
Access control	Access-control policy inferred from normal interaction of client	Insert authorization checks needed to enforce inferred policy.

Fig. 7. Summary of Inferred Specifications and Repairs

ACKNOWLEDGMENT

We thank the anonymous reviewers for their comments.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see copyright notice for non-US Government use and distribution. DM-0003963

REFERENCES

[1] R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2009.

[2] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT press, 1999.

[3] Z. Coker and M. Hafiz. Program transformations to fix C integers. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pages 792–801. IEEE Press, 2013.

[4] P. Cousot and R. Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles Of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[5] V. Dallmeier, A. Zeller, and B. Meyer. Generating fixes from object behavior anomalies. In *IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009.

[6] R. B. Dannenberg, W. Dormann, D. Keaton, R. C. Seacord, D. Svoboda, A. Volkovitsky, T. Wilson, and T. Plum. As-if infinitely ranged integer model. In *IEEE International Symposium on Software Reliability Engineering*, 2010.

[7] D. Dhurjati and V. Adve. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE*. ACM, 2006.

[8] G. J. Duck and R. H. Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142. ACM, 2016.

[9] Q. Gao, Y. Xiong, Y. Mi, L. Zhang, W. Yang, Z. Zhou, B. Xie, and H. Mei. Safe memory-leak fixing for C programs. In *ICSE*, 2015.

[10] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *ACM SIGPLAN Notices*, volume 37, pages 69–82. ACM, 2002.

[11] N. Hasabnis, A. Misra, and R. Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144. ACM, 2012.

[12] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *USENIX Annual Technical Conference, General Track*, 2002.

[13] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit. Automated atomicity-violation fixing. In *PLDI*, 2011.

[14] L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *International Conference on Computer Aided Verification (CAV)*, 2013.

[15] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 2012.

[16] F. Logozzo and T. Ball. Modular and verified automatic program repair. In *OOPSLA 2012*.

[17] M. Monperrus. Automatic Software Repair: a Bibliography. Technical Report hal-01206501, University of Lille, Nov 2015. http://www.monperrus.net/martin/survey-automatic-repair.pdf.

[18] P. Muntean, V. Kommanapalli, A. Ibing, and C. Eckert. Automated Generation of Buffer Overflow Quick Fixes Using Symbolic Execution and SMT. In *International Conference on Computer Safety, Reliability, and Security*, pages 441–456. Springer, 2015.

[19] D. Muthukumaran, D. O'Keeffe, C. Priebe, D. Eyers, B. Shand, and P. Pietzuch. FlowWatcher: Defending against data disclosure vulnerabilities in web applications. In *ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[20] S. Nagarakatte, M. M. Martin, and S. Zdancewic. Everything You Want to Know About Pointer-Based Checking. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 32. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[21] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: highly compatible and complete spatial memory safety for C. In *PLDI 2009*, 2009.

[22] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *POPL*. ACM, 2002.

[23] R. Rugina and M. C. Rinard. Symbolic bounds analysis of pointers, array indices, and accessed memory regions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(2):185–235, 2005.

[24] M. Schordan and D. J. Quinlan. A Source-to-Source Architecture for User-Defined Optimizations. In *Joint Modular Languages Conference (JMLC)*, 2003.

[25] A. Shaw, D. Doggett, and M. Hafiz. Automatically Fixing C Buffer Overflows Using Program Transformations. In *Dependable Systems and Networks (DSN)*. IEEE, 2014.

[26] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis. Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News*, 2009.

[27] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference*. USENIX Association, 2005.

[28] M. S. Simpson and R. K. Barua. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.

[29] S. Son, K. S. McKinley, and V. Shmatikov. Fix Me Up: Repairing Access-Control Bugs in Web Applications. In *Network and Distributed System Security Symposium (NDSS) 2013*.

[30] S. Son, K. S. McKinley, and V. Shmatikov. RoleCast: Finding Missing Security Checks when You Do Not Know What Checks Are. In *OOPSLA '11*. ACM, 2011.

[31] D. Tarditi. Extending C with bounds safety. https://github.com/Microsoft/checkedc/releases/download/v0.5-final/checkedc-v0.5.pdf, June 2016.

[32] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek. Improving Integer Security for Systems with KINT. In *OSDI*, 2012.

[33] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In *Computer Security–ESORICS 2010*. Springer, 2010.