Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2015-018 May 26, 2015

Automatic Discovery and Patching of Buffer and Integer Overflow Errors

Stelios Sidiroglou-Douskos, Eric Lahtinen, and Martin Rinard

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Automatic Discovery and Patching of Buffer and Integer Overflow Errors

Stelios Sidiroglou-Douskos, Eric Lahtinen, and Martin Rinard
MIT EECS & CSAIL
{stelios, elahtinen, rinard}@csail.mit.edu

ABSTRACT

We present Targeted Automatic Patching (TAP), an automatic buffer and integer overflow discovery and patching system. Starting with an application and a seed input that the application processes correctly, TAP dynamically analyzes the execution of the application to locate target memory allocation sites and statements that access dynamically or statically allocated blocks of memory. It then uses targeted error-discovery techniques to automatically generate inputs that trigger integer and/or buffer overflows at the target sites.

When it discovers a buffer or integer overflow error, TAP automatically matches and applies patch templates to generate patches that eliminate the error. Our experimental results show that TAP successfully discovers and patches two buffer and six integer overflow errors in six real-world applications.

1. INTRODUCTION

Integer and buffer overflow errors are a well-known source of serious security vulnerabilities. We present a new system, Targeted Automatic Patching (TAP), that can automatically discover and patch both integer and buffer overflow errors.

Starting with a seed input that the application can process successfully, TAP uses targeted error discovery to automatically generate inputs that trigger either 1) buffer overflows at statements that access dynamically or statically allocated blocks of memory or 2) integer overflows in the computations of expressions that specify the sizes of allocated memory blocks. Once TAP obtains such an error-triggering input, it matches a collection of templates against expressions that characterize the source of the error. If one of the templates matches, TAP uses the template to automatically generate and insert a source-level patch that checks for the error and exits if the input would trigger the error. Given appropriate binary or binary hot patching infrastructure, it would be straightforward to patch the (potentially running) binary as well.

Because the generated patches check a condition and, if the condition holds, exit the program, they do not introduce new vulnerabilities. They instead simply narrow the set of inputs that the application chooses to process. We also discuss templates that are designed to enhance the survival in the face of buffer overflow errors. These templates apply either failure-oblivious computing [40] (discarding out of bounds writes, manufacturing out of bounds reads) or boundless memory blocks [42] (storing out of bounds writes in a hash table to return for corresponding out of bounds reads). The goal is to enable the program to execute through out of bounds accesses without corruption so that it can continue on to deliver useful service and results. Server programs (which process a sequence of inputs) are a particularly appropriate class of programs for these kinds of patches — they often enable the server to survive errors triggered by one input so that the server can continue on to process subsequent inputs correctly.

This paper makes the following contributions:

- **Template-Based Patching:** It introduces templates that TAP uses to generate patches that eliminate buffer and integer overflow errors. These templates operate with symbolic expressions that characterize the errors in the applications. An automatic patch insertion algorithm takes the generated patches and inserts them into the application to check for the buffer or integer overflow condition. If the condition holds, the patch exits the application before the overflow occurs.
- **Buffer Overflow Discovery Algorithm:** It presents a new buffer overflow discovery algorithm. This algorithm leverages information present in implicit flows to find and exploit incorrectly coded checks that are designed to (but in fact do not) protect against buffer overflows. It then leverages this information to automatically generate inputs that trigger the buffer overflow error.
- **Experimental Results:** It presents results from experiments that run TAP on six real-world applications. These results show that TAP can successfully discover and patch two buffer and six integer overflow errors in these six applications.

2. EXAMPLE

We next present an example that illustrates how TAP discovers and patches a buffer overflow error. Figure 1 presents a simplified example from gif2tiff 4.0.3 [6]. This example reads `datasize` from the input file, then uses `datasize` to compute how many entries in the statically allocated `prefix` file to clear. A large `datasize` field can trigger a buffer overflow at the `prefix[code] = 0` statement.

2.1 Buffer Overflow Discovery

TAP first performs an instrumented execution of the gif2tiff application on a benign seed input file that gif2tiff can process successfully (such inputs are typically available, for example, as part of standard regression test suites). This instrumented execution operates on the compiled binary representation of the program. It uses Valgrind to obtain and analyze the sequence of executed instructions, analyzing the execution at the level of the Valgrind VEX IR [36]. This first instrumented execution records which input bytes directly influence the value of each computed expression. It also records control dependence information between conditional branches and corresponding control dependent statements whose values are influenced by overlapping sets of input bytes.

In the example, this instrumented execution records the fact that the statement `prefix[code] = 0` is control dependent on the loop condition `code < clear`. It also records the fact that the loop condition is influenced by `code`, which also influences the

```
unsigned int prefix[4096];
int datasize;
int clear;

readraster(void)
{
  register int code;

  datasize = getc(infile);
  clear = 1 << datasize;

  for (code = 0; code < clear; code++) {
    prefix[code] = 0;
  }
}
```

Figure 1: (Simplified) gif2tiff Buffer Overflow Error

```
readraster(void)
{
  register int code;

  datasize = getc(infile);
  clear = 1 << datasize;

  for (code = 0; code < clear; code++) {
    if ((int64_t) code << 2 >= 16384)
    { exit(-1); }
    prefix[code] = 0;
  }
}
```

Figure 2: Patched gif2tiff Code

expression used to index the array. Finally, it records the fact that the value of `clear` is influenced by input file bytes, but the size of the `prefix` array is not.

TAP therefore proceeds under the hypothesis that the loop condition is designed to prevent any buffer overflow at the statement `prefix[code] = 0`. But because the loop condition involves input bytes that do not influence the size of the allocated `prefix` array, it may be possible to create an input file that causes gif2tiff to generate a value for `code` that 1) satisfies the loop condition check but 2) is larger than the size of the indexed array and therefore generates an overflow.

TAP next executes another instrumented exection of gif2tiff. This instrumented execution computes symbolic expressions that capture the complete computation of relevant values from constants and input bytes. This execution determines that, for the seed input, `datasize` is byte 415 of the input. It also determines that the value of `clear` is `1 << Byte(415)` (here `Byte(415)` is byte 415 of the input) and that the offset used to index the `prefix` array is (at the analyzed VEX IR level) `clear << 2`. TAP uses this value to generate the following two constraints (here 16384 is the size of the `prefix` array in bytes):

$$(\texttt{code} << 2) \geq 16384 \qquad \texttt{code} < (1 << \texttt{Byte}(415))$$

The first constraint forces the solver to produce an offset that generates an overflow, while the second constraint ensures that the solver produces values for the index and input bytes that together satisfy the loop condition.

TAP then invokes the Z3 SMT solver [20], which processes the two constraints to produce a value of 30 for input byte 415. Starting with the seed file, TAP generates a new gif file with 30 at byte 415.[1] TAP then runs gif2tiff on the new input file and verifies that the input file triggers the buffer overflow.

2.2 Patch Generation

TAP next moves on to generate a patch that checks for a buffer overflow and, if the check detects an overflow, exit the application. Conceptually, the patch checks the index against the size of the (dynamically or statically) allocated buffer, then exits the application if the index exceeds the buffer size. In our example, TAP generates the following patch (appropriately adjusting the value of `code` and the size of the `prefix` array):

```
if ((int64_t) code << 2 >= 16384) { exit(-1); }
```

TAP inserts the patch before the `prefix[code] = 0` statement, ensuring that the application exits before the overflow occurs. Figure 2 presents the patched code.

2.3 Additional Complexity

Of course, TAP can successfully discover and patch much more complex errors. Additional sources of complexity include dynamically allocated buffers, much more complex index expressions, loop and if conditions that are designed to check for overflows, relevant code that is spread arbitrarily throughout the application, low-level complexity associated with instrumenting binaries to record symbolic expressions for relevant values, and finding an appropriate insertion point for patches that need to access input bytes scattered throughout the memory of the application. And of course TAP can also discover and patch integer overflow errors. See Sections 3 and 4.

3. ERROR DISCOVERY ALGORITHMS

The TAP integer and buffer overflow discovery algorithms both start with an application and at least one seed input (which the application may process successfully without error). Based on instrumented executions of the application running the seed input(s), they both use a goal-directed approach to generate a new error-triggering input that exposes the buffer and/or integer overflow error in the application. In addition to the error-triggering input, the discovery algorithms also produce symbolic expressions that the patch generation algorithms use to generate patches that eliminate the errors.

3.1 Discovering Buffer Overflow Errors

We illustrate the buffer overflow discovery algorithm with the following example:

```
char *buf = malloc(size);
...
for (i = 0; i < height*width; i++) {
  buf[i] = 0;
}
```

Here `size` is the size field of an image input file, `height` is the height field of the image, and `width` is the width field of the image. An input file in which `size < height*width` will trigger a buffer overflow at the statement `buf[i] = 0`. The key concept here is that the check `height*width` is designed to prevent a buffer overflow, but because the size of the allocated buffer depends on `size` but not `height` or `width`, there are inputs for which the check does not prevent the overflow. The TAP buffer overflow discovery algorithm is designed to find (generalizations of) such code

[1] For gif files TAP can simply rewrite the relevant bytes directly. For more complex file formats TAP uses a combination of Hachoir [3] and Peach [7] to generate new input files.

patterns and generate inputs that expose any buffer overflow errors present in the patterns.

The TAP buffer overflow discovery algorithm therefore dynamically monitors the execution of the application on the seed input to find code patterns of the following form (here the `if` may be any conditional branch, including conditional branches from loop conditions and if statements):

```
p = malloc(s);
...
if (e op b) {
  ...
  *(p + o)
  ...
}
```

Here `s`, `e`, `b`, and `o` are expressions that TAP derives during the monitoring process. `op` is a comparison operator (such as <, >, <=, =<, ==, !=, ...). Note that the target memory access `*(p + o)` is control dependent on the condition `e op b` so that there is an implicit flow between the condition and the target access. We note that there is no requirement that the allocation, condition, or target access be located in the same procedure or module — because TAP dynamically monitors the execution to track the flow of values through the program, it can find overflows in which these three elements are located in distant parts of the program.

TAP also works with patterns that statically allocate the buffer. In this case `s` is simply the constant that is the size of the statically allocated buffer.

3.2 Target Patterns

TAP operates under the principle that the condition `e op b` may be designed to check that the offset `o` is within the bounds of the allocated block of memory `p`, but that the check may be incorrect. In particular, if the bound `b` is influenced by more input fields than the size `s` of the allocated memory block `p`, then it may be possible to generate an input that satisfies the condition but overflows the target access. If the condition was (incorrectly) designed to prevent an overflow, then the condition `e op b` will have some relationship with the offset `o`. Specifically, TAP checks for patterns in which the following two constraints hold:

$$IB_b - IB_s \neq \emptyset \qquad V_e \cap V_o \neq \emptyset$$

Here IB_b is the set of input file bytes that influence the value of `b`, IB_s is the set of input bytes that influence the value of `s` (for statically allocated buffers $IB_s = \emptyset$), V_e is the set of variables in `e`, and V_o is the set of variables in `o`.

The constraint $IB_b - IB_s \neq \emptyset$ checks that the condition depends on some input bytes that are not involved in the computation of the size of the allocated buffer. The constraint $V_e \cap V_o \neq \emptyset$ checks that there is some relationship between the condition and the access offset `o`.

The TAP monitoring system obtains the information required to compute IB_b, IB_s, V_e, and V_o by tracking how input fields and values flow through the application as it runs [49].

3.3 Generated Constraint System

When TAP finds such a pattern, it attempts to generate a new input that triggers the overflow. Specifically, it generates and solves (using the Z3 SMT solver [20]) the following two constraints:

$$\texttt{o} \geq IE_s \qquad \texttt{e op } IE_b$$

Here IE_s is a symbolic expression (the variables in this symbolic expression represent input bytes) that captures the complete computation of `s` starting from constants and input bytes, and similarly for IE_b. The TAP monitoring system produces these expressions as part of its instrumented execution of the program [49]. Here the constraint $o \geq IE_s$ forces the solver to generate a solution that causes the offset `o` to exceed the size `s` of the allocated buffer `p`. The constraint e `op` IE_b ensures that the solution also satisfies the condition so that the check does not skip the out of bounds access.

The solution to this constraint system delivers a set of values for the input bytes involved in the computation of `s` and `b`. These input byte values satisfy these constraints.

The constraint system in the illustrative example above contains the following two constraints:

$$\texttt{i} \geq IE_{size} \qquad \texttt{i} < IE_{height} * IE_{width}$$

Note that `i`, by linking the condition to the values in the target access, makes the constraints associated with the implicit flow explicitly present in the constraint system. This is a critical step in enabling the solver to find input byte values that trigger the overflow.

Given a solution to the constraint system, TAP next generates a new input file. This file replaces the old values in the seed input file with the generated input byte values from the solver. If this new file causes the application to follow a path through the condition to the access, it will trigger the overflow.

3.4 Conditional Branch Enforcement

In some cases, however, the application may contain other checks that prevent the application from following a path that triggers the overflow. In this case TAP uses the goal-directed conditional branch enforcement algorithm from DIODE [49] to find input byte values that cause the application to traverse a path that executes the target access. The enforcement algorithm includes not just the TAP constraints above, but other constraints generated by DIODE designed to force the flow of control to the target access. If successful, this algorithm produces an input file that triggers the buffer overflow error.

3.5 Discovering Integer Overflow Errors

The TAP integer overflow discovery algorithm is taken directly from DIODE [49]. This algorithm monitors the execution of the application to identify memory allocation site and construct symbolic expressions that capture the size of the allocated buffer as a function of the input bytes. It then uses goal-directed conditional branch enforcement to generate inputs that 1) overflow the computation of the size of the allocated buffer while 2) forcing the application to take a path that executes the statement at the memory allocation site.

4. PATCH GENERATION

We next discuss the TAP patch generation algorithms. These algorithms start with symbolic expressions generated by the instrumented executions of the application that TAP performs to obtain error-triggering inputs. It is also straightforward to start with any error-triggering input obtained by any means whatsoever, then use the TAP instrumented execution functionality to obtain the required symbolic expressions. TAP contains a set of templates that it matches against the symbolic expressions. If a template matches, TAP applies the template to generate an associated patch and inserts the patch into the application.

4.1 Buffer Overflow Patches

The TAP buffer overflow patch generation algorithm starts with the two expressions IE_s and o produced by the TAP monitoring system. Conceptually, the generated patch should check if $o >= IE_s$ and, if so, exit to avoid the overflow. Recall that the variables in the expression IE_s represent input field bytes, not program variables. To generate a source-level patch that implements this expression,

TAP must translate IE_s into the name space of the application. In other words, TAP must find a place in the application where the input byte values are accessible via program variables and source-level expressions in the application. It can then replace the input bytes in IE_s with source-level expressions that contain the corresponding input bytes.

4.2 Patch Insertion

TAP performs the patch insertion using the CodePhage patch insertion algorithm [51]. This algorithm tracks the flow of input bytes through the variables and memory of the program. Given an expression over the input bytes and a candidate insertion point in the program, the insertion algorithm traverses the heap to find source-level expressions that contain the relevant input bytes [51]. These source-level expressions may be simply program variables (if the program read the relevant input bytes into a program variable) or more complex expressions that start at a program variable, then traverse the heap to access a memory location that contains the input bytes. If expressions for all of the required input bytes are available at the candidate insertion point, the patch insertion algorithm uses the discovered source-level names for the input bytes to translate the expression into the source-level name space of the application at the candidate insertion point.

The patch insertion algorithm also ensures that the patch is inserted to execute before the target access. This ensures that the patch terminates the program before the overflow at the target access occurs.

4.3 Buffer Overflow Patch Template

Given the expression IE_s and translation $TR(IE_s)$ into the source-level name space of the program at the candidate insertion point, the TAP buffer overflow patch template generates the following patch at the candidate insertion point:

```
if (o >= TR(IE_s)) { exit (-1); }
```

This patch checks to see if the target access will overflow the buffer via the offset `o` and, if so, exits the program to prevent the overflow from occurring.

It is also possible to apply a failure-oblivious computing [40] patch template. Instead of exiting when the target access will overflow the buffer, it instead discards out of bounds writes and manufactures values (zero, one, a predetermined sequence of values, or sequence of random values) to return as the result of out of bounds reads. It is also possible to apply a boundless memory blocks [42] patch template that stores out of bounds writes in a hash table for subsequent out of bounds reads to read. The goal of both of these patch templates is to enable successful continued execution despite the out of bounds access.

4.4 Patch Validation

TAP next runs the patched application on the error-triggering input to verify that the patch eliminates the overflow. It also runs the patched application on any available input test suite to check that the patch does not interfere with correct executions on inputs that the unpatched program processes correctly.

4.5 Integer Overflow Patches

The TAP integer overflow patch generation algorithm starts with an expression IE_s and a memory allocation site that allocates a buffer of size `s`. The DIODE integer overflow discovery algorithm generates this expression. This expression captures the complete calculation that the application performs to compute the size `s` of the allocated buffer. IE_s is an expression over the input bytes.

TAP starts with the expression IE_s and generates an overflow detection expression D that checks for an integer overflow in the computation of the size `s` [31]. These overflow detection expressions often take the form of templates that apply to certain computation patterns. Here are several overflow detection expressions that TAP currently implements:

- **Unsigned 32 Bit Multiplication Overflow:** If IE_s is of the form $A * B$, where A and B represent unsigned 32 bit integers, the overflow detection expression D is:

  ```
  ((uint64_t)A * (uint64_t)B > MAX_UINT32)
  ```

  We note that because the detection expression D captures the entire computation of s, the generated overflow check can detect overflows that occur even if the computation of $A * B$ occurs very far away in the program from allocation site.

- **Signed 32 Bit Multiplication Overflow:** If IE_s is of the form $A * B$, where A and B represent signed 32 bit integers, the overflow detection expression D is:

  ```
  ((int64_t)A * (int64_t)B < MIN_INT32) ||
  ((int64_t)A * (int64_t)B > MAX_INT32)
  ```

  As for unsigned overflows, because the detection expression D captures the entire computation of s, the generated overflow check can detect overflows that occur even if the computation of $A * B$ occurs very far away in the program from allocation site.

- **64 to 32 Bit Cast Overflow:** If IE_s is of the form $(C)A$, where A is a 64-bit unsigned integer and the cast C converts A to a 32-bit unsigned integer, the overflow detection expression D is `((uint64_t)((uint32_t)A)) != A`. We note that because the detection expression D captures the entire computation of s, the generated overflow check can detect overflows that occur even if the cast occurs very far away in the program from allocation site.

More generally, it is possible to apply a general integer overflow patch template that, given an arbitrary symbolic expression IE_s, automatically generates a patch that checks if any subcomputation of IE_s overflows. Because IE_s captures the complete computation of the size of the allocated memory block starting from constants and input bytes, this patch can detect overflows that occur in subcomputations arbitrarily far away (in the code or computation) from the allocation site `s`.

4.6 Patch Insertion

As for buffer overflow patches, TAP uses the CodePhage patch insertion algorithm to find an appropriate patch insertion point and translate the detection expression D into the name space of the application at the patch insertion point.

The patch insertion algorithm also ensures that the patch is inserted to execute before the memory allocation site whose size computation produces the overflow.

4.7 Integer Overflow Patch Template

The TAP integer overflow patch template generates the following patch at the candidate insertion point:

```
if (TR(D)) { exit (-1); }
```

where $TR(D)$ is the translation of D at the patch insertion point. The CodePhage patch insertion algorithm generates this translation [51].

Application	Discovered Overflow Location	Overflow Type	Patch Template	Patch Insertion Point	Number of Candidate Insertion Points	Check Size
CWebP 0.3.1	jpegdec.c:248	Integer	$A * B$	jpegdec.c:246	23-2-16 = 5	$45 \rightarrow 5$
Dillo 2.1	png.c:203	Integer	$A * B$	png.c:195	16-0-10 = 6	$46 \rightarrow 4$
Display 6.5.2	xwindow.c:5619	Integer	$A * B$	xwindow.c:5590	71-4-57 = 10	$19 \rightarrow 5$
Display 6.5.2	cache.c:3717	Integer	$A * B$	cache.c:3788	48-4-41 = 3	$24 \rightarrow 8$
SwfPlay 0.5.5	jpeg_rgb_decoder.c:253	Integer	$A * B$	jpeg_rgb_decoder.c:252	30-2-24 = 4	$13 \rightarrow 5$
SwfPlay 0.5.5	jpeg.c:192	Integer	$(C)A$	jpeg.c:186	26-0-25 = 1	$546 \rightarrow 22$
JasPer 1.9	jpc_dec.c:500	Buffer	$\circ \geq s$	jpc_dec.c:492	32-0-31 = 1	$44 \rightarrow 10$
gif2tiff 4.0.3	gif2tiff.c:355	Buffer	$\circ \geq s$	gif2tiff.c:342	1-0-0 = 1	$2 \rightarrow 2$

Figure 3: Summary of TAP Experimental Results

Application	Discovered Overflow Location	Seed Input Analysis Time	Error Discovery Time	Patch Generation and Validation Time
CWebP 0.3.1	jpegdec.c:248	4 min 13 sec	0 min 10 sec	0 min 48 sec
Dillo 2.1	png.c@203	34 min 59 sec	2 min 54 sec	1 min 40 sec
Display 6.5.2	xwindow.c@5619	7 min 3 sec	0 min 1 sec	2 min 49 sec
Display 6.5.2	display.c@4393	7 min 3 sec	0 min 3 sec	2 min 54 sec
SwfPlay 0.5.5	jpeg_rgb_decoder.c:253	23 min 29 sec	16 min 9 sec	2 min 0 sec
SwfPlay 0.5.5	jpeg.c:192	23 min 29 sec	26 min 0 sec	0 min 1 sec
JasPer 1.9	jpc_dec.c:500	11 min 0 sec	0 min 11 sec	0 min 28 sec
gif2tiff 4.0.3	gif2tiff.c:355	13 min 41 sec	0 min 1 sec	0 min 24 sec

Figure 4: TAP Seed Input Analysis, Error Discovery, and Patch Generation and Validation Times

4.8 Patch Validation

As for buffer overflow patches, TAP verifies that the generated patch 1) enables the application to correctly reject the error-triggering input and 2) does not interfere with the semantics of test suite inputs that the unpatched program already processes correctly. TAP also leverages the semantics of integer overflow errors to perform an additional validation step [51]. Specifically, TAP analyzes the generated patch, the expression that overflows, and other existing checks that are relevant to the error to verify that there is no input that 1) satisfies the checks to traverse the exercised path through the program to the overflow and also 2) triggers the overflow.

5. EXPERIMENTAL RESULTS

We implemented the TAP buffer and integer overflow discovery and patching algorithms and applied these algorithms to discover and patch a set of buffer and integer overflow errors in the following set of benchmark applications: JasPer 1.9 [5], gif2tiff 4.0.3 [6], CWebP 0.31 [1], Dillo 2.1 [2], swfplay 0.55 [10], and Display 6.5.2-8 [4]. TAP discovered and patched buffer overflow errors in JasPer 1.9 and gif2tiff 4.0.3 and integer overflow errors in CWebP 0.31, Dillo 2.1, swfplay 0.55, and Display 6.5.2-8. The buffer overflow and two of the integer overflow errors are listed in the CVE database. One of the integer overflow errors was first discovered by BuzzFuzz [24]; the other four were, to the best of our knowledge, first discovered by DIODE [49]. The errors are triggered by JPG image files (CWebP), PNG image files (Dillo), SWF video files (swfplay), and TIFF image files (Display).

Figure 3 summarizes the results. There is a row in the table for each discovered error. The first column (Application) identifies the application that contains the error. The second column (Discovered Error Location) identifies the source code file and line where the discovered overflow error occurs. The third column (Overflow Type) specifies whether the error was a buffer or integer overflow error. The JasPer buffer overflow error involves a dynamically allocated buffer, while the gif2tiff buffer is statically allocated.

The fourth column (Patch Template) specifies whether the patch template was a multipy template ($A * B$), a cast template $(C)A$, or the buffer overflow template ($\circ \geq s$). The fifth column (Patch Insertion Point) identifies the source code file and line where TAP inserted the generated patch. TAP usually (CWebP, Dillo, SwfPlay, JasPer, gif2tiff) inserts the patch relatively close to the error.

The sixth column (Candidate Insertion Points) contains entries of the form $X - Y - Z = W$. Here X is the number of candidate insertion points, Y is the number of unstable points (TAP filters these points), Z is the number of insertion points at which TAP was unable to translate the patch because it was unable to find source-level expressions for some of the required input bytes, and W is the number of points at which TAP is able to insert a successfully translated patch. See [51] for more details.

The seventh column (Check Size) contains entries of the form $X \rightarrow Y$. Here X is the number of operations in the untranslated representation of the check. Y is the number of operations in the translated check as it is inserted into the recipient. We attribute the significant size reduction to the ability of the CodePhage Rewrite algorithm to recognize complex expressions that are semantically equivalent. The typical scenario is that TAP recognizes that a complex expression containing shifts and masks over input byte values from (for example) endianess conversions as the input byte values are read into the application is equivalent to a single variable or data structure field in the recipient. See [51] for more details.

Figure 4 presents the amount of time TAP requires to perform various activities for each error. The third column (Seed Input Analysis Time) presents the amount of time that TAP requires to perform the instrumented executions of the application on the seed inputs. These instrumented executions record the input bytes that influence each value and generate the required symbolic expressions (IE_s, IE_b) that TAP records to perform the subsequent error discovery and patch phases. In general, these instrumented executions are the most time-consuming part of the end to end TAP system. Dillo and SwfPlay take longer because they are interactive applications. TAP must therefore

timeout the execution instead of simply observing the application terminate when it finishes processing the seed input.

The fourth column (Error Discovery Time) presents the amount of time that TAP takes to execute the error discovery algorithm once it has the information from the instrumented seed input executions. These times are at most several minutes for our set of benchmark applications. The final column (Patch Generation and Validation Time) is the time required to generate and validate the patch. Again, the times are at most several minutes, with the majority of the time devoted to rebuilding the application once TAP has installed the patch.

Overall, the end to end analysis, discovery, and patch generation and validation are always less than 45 minutes and in many cases are much less than 45 minutes. We next discuss some of the specific patches that TAP generates.

5.1 JasPer 1.9

JasPer 1.9 is an open-source image viewing and image processing utility. It is specifically known for its implementation of the JPEG-2000 standard. JPEG-2000 images may be composed of multiple tiles, with the number of tiles specified by a 16 bit field in the input file. JasPer contains an off-by-one error in the code that processes JPEG-2000 tiles. When JasPer processes the tiles, it includes code that is designed to check that the number of tiles actually present in the image is less than or equal to the number specified in the input file. Unfortunately, the check was miscoded — at jpc_dec.c:492, JasPer checks if the number of the current tile is greater than (>) the specified number of tiles. The correct check is a greater than or equal to (>=) check. The result is that JasPer can access tile data beyond the end of the buffer allocated to hold that data.

The following code fragment contains the error. The incorrect check occurs in the first line. The resulting out of bounds access occurs when the application evaluates `tile->partno`:

```
if (JAS_CAST(int, sot->tileno) > dec->numtiles) {
  jas_eprintf("invalid tile number ...\n");
  return -1;
}
/* Set the current tile. */
dec->curtile = &dec->tiles[sot->tileno];
tile = dec->curtile;
/* Ensure that this is the expected part number. */
if (sot->partno != tile->partno) {
  return -1;
}
```

TAP applies the buffer overflow template to obtain the following patch, which it inserts at jpc_dec.c:192 just before the code fragment above:

```
if (((((sot->tileno << 3) << 3) -
    (sot->tileno << 3)) + 40 >=
    ((dec->numtiles << 3) << 3) -
    (dec->numtiles << 3)))
  {exit(-1);}
```

This patch uses shift operations to efficiently compute a multiply by 56 (the size of a single element of the `dec->tiles` array). TAP uses this efficient form because the expressions that it works with are derived from the compiled application and therefore reflect the optimizations that the compiler applies to array indexing operations. 40 is the offset of the `partno` field within an element of the `dec->tiles` array. The check determines whether the accessed offset is larger than the size of the allocated array. Note that to generate this check, TAP must determine that `dec->numtiles` contains the number of elements in the dynamically allocated `dec->tiles` array.

5.2 Dillo 2.1

Dillo is a lightweight graphical web browser. Dillo 2.1 is vulnerable to an integer overflow when decoding the PNG file format. Dillo computes the size as a 32-bit product of the image width, height, and pixel depth from the input file. A check for potentially malicious width and height values is present, but the check itself is unfortunately vulnerable to an overflow. When the buffer size calculation overflows, the allocation at png.c:203 returns a buffer that is too small to hold the decompressed image (CVE-2009-2294).

The potentially malicious width and height check (which is itself vulnerable to an overflow) appears starting at png.c:142 as follows:

```
/* check max image size */
if (abs(png->width*png->height) >
    IMAGE_MAX_W * IMAGE_MAX_H) {
   MSG("Png_datainfo_callback: ... %ldx%ld\n",
       png->width, png->height);
   Png_error_handling(png_ptr, "Aborting...");
   return; /* not reached */
}
```

The integer overflow occurs at png.c:203 as follows when Dillo calculates the size of the buffer to allocate:

```
png->image_data = (uchar_t *)
  dMalloc(png->rowbytes * png->height);
```

Here `png->height` contains the height field from the input png file and `png->rowbytes` is the number of bytes in each row of the image (which is a function of the width field from the input png file).

TAP applies the multiply integer overflow patch template (the $A * B$ template) at png.c:195 as follows:

```
if (((uint64_t) png->rowbytes *
     (uint64_t) png->height >
     __UINT64_C(4294967295))) {exit(-1);}
```

This example illustrates the difficulty that developers can have writing correct code. Even though the developer anticipated the possibility of inputs with potentially malicious width and height fields, the application is still vulnerable to an integer overflow involving those fields.

The remaining integer overflow errors with the $A * B$ template have similar patches.

5.3 Display 6.5.2

ImageMagick Display is an image viewing and formatting utility released as part of the popular ImageMagick suite. Display 6.5.2 contains overflow errors when creating a resized version of the image for display within the GUI window, and when creating a cache buffer for the image during TIFF decompression (a request for pixel space at tiff.c:1044 eventually results in an allocation at cache.c:3717). When the computation of any of these buffer sizes overflows, the allocated memory blocks are too small, causing Display to write beyond the end of the block.

Here is the code at cache.c:3717 that allocates the buffer and contains the overflow:

```
cache_info->pixels=(PixelPacket *)
  AcquireMagickMemory((size_t) cache_info->length);
```

In this code `cache_info->length` is a 64 bit integer containing the number of bytes required to contain the image. Display casts this 64 bit integer to a 32 bit integer before passing it to `AcquireMagickMemory()`. For large image sizes, the overflow occurs at this cast.

The following code, which occurs starting at cache.c:3787, computes `cache_info->length`. All of the calculations are performed in 64 bit arithmetic.

```
cache_info->rows=image->rows;
cache_info->columns=image->columns;
...
number_pixels=(MagickSizeType)
  cache_info->columns*cache_info->rows;
...
length=number_pixels*packet_size;
...
cache_info->length=length;
```

Here `image->rows` contains the bytes from the input file that specify the number of rows in the image, `image->columns` contains the bytes from the input file that specify the number of columns in the image, and `packet_size` is the constant 8.

TAP generates the following patch, which it inserts just before the calculations presented above starting at cache.c:3787:

```
if ((8 * (uint64_t) ((uint32_t)
    ((uint64_t) image->rows *
     (uint64_t) image->columns)) !=
     (uint64_t) ((uint32_t)
     (8 * (uint64_t) ((uint32_t)
     ((uint64_t) image->rows *
     (uint64_t) image->columns))))))
  {exit(-1);}
```

Here TAP recognizes that `cache_info->length` is the product of the number of rows and columns from the input file times the packet size (8). The check converts the value to 32 bits, then back to 64 bits, and checks if the conversion does not produce the original value.

6. RELATED WORK

We discuss related work in buffer and integer overflow discovery as well as work in automatic error patching (see [49, 51, 31]).

Random and Directed Fuzzing: Random fuzzing has been shown to be surprisingly effective in uncovering errors [33, 52] and is heavily used by security researchers [9, 7, 43]. But because most randomly generated inputs fail input sanity checks, random fuzzing has been relatively ineffective at generating inputs that trigger errors (such as integer overflows) deep inside applications. Its ability to generate such inputs can be especially limited for programs that process deeply structured formats such as videos.

Motivated by the need to expose errors deep inside applications, researchers have proposed directed fuzzing techniques [24, 54, 23]. BuzzFuzz [24] and TaintScope [54] use taint tracking to identify input bytes that influence values at critical program sites such as memory allocation sites and system calls. In contrast with random fuzzing techniques that modify the entire input, these techniques then fuzz only the input bytes that influence critical program points. While successful at reducing the size of the fuzzed input space, our results indicates that these directed techniques are ineffective at finding the carefully crafted inputs required to navigate the sanity checks and expose integer overflow errors. Because these directed fuzzing systems operate directly on the raw binary input bytes, the modifications can also produce syntactically incorrect inputs that immediately fail the sanity checks.

Symbolic Test Generation: Symbolic test generation (i.e., concolic testing) has been proposed as an alternative to random and directed fuzzing [46, 25, 26, 34, 15, 14, 53, 27]. These systems execute programs both concretely and symbolically on a seed input until an interesting program expression is reached (e.g., an assert, a conditional or a specific expression). Although successful in many cases [26, 14, 15, 34], symbolic test generation faces several challenges [47, 13]. Specifically, once past the initial parsing stages, the resulting deeper program paths may produce very large constraints with complex conditions that are beyond the capabilities of current state of the art constraint solvers. Our results also show that the path taken by a seed input may contain additional blocking checks that can prevent a constraint solver from generating inputs that satisfy the checks and trigger an overflow [49].

SmartFuzz [34] is a symbolic test generation tool to discover integer overflows, non-value-preserving width conversions, and potentially dangerous signed/unsigned conversions. SmartFuzz, like other concolic systems, is limited by deep program paths and blocking checks.

Dowser [27] is a fuzzer that combines taint tracking, program analysis, and symbolic execution to find buffer overflows. The key idea is to use program analysis to guide symbolic execution (e.g.,KLEE [14]) along a path that is more likely to discover buffer overflows than running symbolic execution over the entire program. Like most concolic systems, Dowser optimizes for path coverage and is thus unlikely to discover integer overflow errors.

TAP uses DIODE [49] to discover integer overflow errors. DIODE differs from these echniques in that it is *targeted* — instead of exploring paths to find critical sites, it starts with a critical site that is executed by a seed input, then uses a variety of techniques that are designed to produce inputs that successfully navigate sanity and blocking checks to trigger an overflow at the critical site. TAP extends the targeted DIODE approach to discover buffer overflow errors.

Runtime and Library Support: To alleviate the problem of false positives, several research projects have focused on runtime detection tools that dynamically insert runtime checks before integer operations [12, 56, 22]. Another technique is to use safe integer libraries such as SafeInt [8] and CERT's IntegerLib [45] to perform sanity checks at runtime. Using these libraries requires developers to rewrite existing code to use safe versions of integer operations. TAP, in contrast, pro-actively finds integer errors during testing (i.e., does not rely on observing a malicious input in the wild) and imposes no runtime overhead other than the inserted patch.

Input Rectification is another technique that can protect applications from integer overflow errors [41, 29, 30] by empirically learning input constraints from benign training inputs and then monitoring inputs for violations of the learned constraints. Instead of discarding inputs that violate the learned constraints, input rectification modifies the input so that it satisfies the constraints. The goal is to nullify potential errors while still enabling the program to successfully process as much input data as possible. Because it learns the constraints from examples, the technique is susceptible to false positives.

Static Analysis For Finding Integer Errors: Several static analysis tools have been proposed to find integer overflow and/or sign errors [55, 18, 44]. KINT [55], for example, analyzes individual procedures, with the developer optionally providing procedure specifications that characterize the value ranges of the parameters. Despite substantial effort, KINT reports a large number of false positives [55]. In contrast, TAP generates inputs that prove the existence of integer errors without any false positives.

SIFT [31] generates input filters that nullify integer overflow errors associated with critical memory allocation and block copy sites. SIFT uses a sound static program analysis to generate filters that discard inputs that may trigger overflow errors. SIFT requires access to source code and is not designed to identify errors. TAP, in contrast, operates directly on stripped x86 binaries with no need for source code to generate overflow-triggering inputs (although it does require source code access to generate and insert the patch).

Runtime Program Repair: Failure-Oblivious Computing enables applications to survive common buffer overflow memory errors [40]. It recompiles the application to discard out of bounds writes, manufacture values for out of bounds reads, and enable applications to continue along their normal execution paths. RCV [32] enables

applications to dynamically recover from divide-by-zero and null-dereference errors. When such an error occurs, RCV attaches the application, applies a fix strategy that typically ignores the offending instruction, forces the application to continue along the normal execution path, contains the error repair effect, and detaches from the application once the repair succeeds. SRS [35] enables server applications to survive memory corruption errors. When such an error occurs, it enters a crash suppression mode to skip any instructions that may access corrupted values. It reverts back to normal mode once the server moves to the next request.

ClearView [38] first learns a set of invariants from training runs. When a learned invariant is violated during the runtime execution, it generates repairs that enforce the violated invariant via binary instrumentation. It is also possible to learn and enforce data structure consistency properties [21].

Jolt [16] and Bolt [28] enable applications to survive infinite loop errors. Bolt attaches to unresponsive applications, detects if the application is in an infinite loop, and if so, either exits the loop or returns out of the enclosing function to enable the application to continue successful execution.

DieHard [11] provides probabilistic memory safety in the presence of memory errors. In stand-alone mode, DieHard replaces the default memory manager with a memory manager that places objects randomly across a heap to reduce the possibility of memory overwrites due to buffer overflows. In replicated mode, DieHard obtains the final output of the application based on the votes of multiple replications. Exterminator [37] automatically generates patches for buffer overflow and dangling pointer errors. Starting with an input that triggers the error, Exterminator patches overflow errors by padding allocated objects and patches dangling pointer errors by deferring object deallocations.

Rx [39] and ARMOR [17] are runtime recovery systems based on periodic checkpoints. When an error occurs, Rx [39] reverts back to a previous checkpoint and makes system-level changes (e.g, thread scheduling, memory allocations, etc.) to search for executions that do not trigger the error. ARMOR [17] reverts back to a previous checkpoint and finds semantically equivalent workarounds for the failed component based on user-provided specifications.

Error Virtualization [50, 48] is a general error recovery technique that retrofits exception-handing capabilities to legacy software. Failures that would otherwise cause a program to crash are turned into transactions that use a program's existing error handling routines to survive unanticipated faults.

All of the above techniques aim to repair the application at runtime to recover from or nullify the error. In contrast, TAP is designed to generate source-level patches that can be compiled into the application to eliminate the error without additional runtime monitoring or recovery support.

Runtime Checks and Library Support: To alleviate the problem of false positives, several research projects have focused on runtime detection tools that dynamically insert runtime checks before integer operations [12, 19, 56, 22]. One drawback is that the inserted checks, which are placed all throughout the application, can impose non-negligible overhead. TAP, in contrast, only inserts checks when it has found an input that causes a buffer or integer overflow. It therefore imposes almost no overhead. TAP also discovers buffer and integer overflow errors (as opposed to simply aspiring to detect overflows).

Another technique is to use safe integer libraries such as SafeInt [8] and CERT's IntegerLib [45] to perform sanity checks at runtime. Using these libraries requires developers to rewrite existing code to use safe versions of integer operations. TAP, in contrast, discovers and patches errors in existing applications without requiring developers to rewrite code.

Benign Integer Overflows: In some cases, developers may intentionally write code that contains benign integer overflows [55, 53, 22]. A potential concern is that techniques that nullify overflows may interfere with the intended behavior of such programs [55, 53, 22]. Because TAP focuses on critical memory allocation sites that are unlikely to have such intentional integer overflows, it is unlikely to nullify benign integer overflows and therefore unlikely interfere with the intended behavior of the program.

CodePhage: Given multiple applications that process the same input files and an input file that triggers an error in one application, CodePhage finds checks in other applications that enable these other applications to successfully process the input file [51]. It then uses multi-application code transfer to transfer these checks into the original application and eliminate the error. TAP differs in that its code patch templates enable it to generate patches in the absence of other applications that 1) can process the same inputs and 2) have checks that eliminate the error.

7. CONCLUSION

Buffer and integer overflow errors are a longstanding and still serious source of security vulnerabilities. TAP uses sophisticated program instrumentation and analysis technology to automatically discover and patch both buffer and integer overflow errors. By proactively discovering errors, TAP can enable organizations to find potentially serious security vulnerabilities before they are exploited. By automatically generating patches with no need to wait for a human developer to analyze the error and produce a (potentially buggy) patch, TAP can dramatically reduce the time between when the vulnerability is discovered and when it is patched. TAP therefore promises to dramatically reduce the time and effort required to discover and eliminate this important class of security vulnerabilities.

8. REFERENCES

[1] Cwebp. `https://developers.google.com/speed/webp/docs/cwebp`.

[2] Dillo. `http://www.dillo.org/`.

[3] Hachoir. `http://bitbucket.org/haypo/hachoir/wiki/Home`.

[4] Imagemagick. `http://www.imagemagick.org/script/index.php`.

[5] The jasper project home page. `http://www.ece.uvic.ca/~frodo/jasper/`.

[6] Libtiff. `http://www.remotesensing.org/libtiff/`.

[7] Peach fuzzing platform. `http://peachfuzzer.com/`.

[8] SafeInt. `http://safeint.codeplex.com/`.

[9] SPIKE fuzzing platform. `http://www.immunitysec.com/resources-freesoftware.shtml`.

[10] Swfdec. `http://swfdec.freedesktop.org/wiki/`.

[11] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06', pages 158–168. ACM, 2006.

[12] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song. RICH: Automatically protecting against integer-based vulnerabilities. *Department of Electrical and Computing Engineering*, page 28, 2007.

[13] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible:

Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143–157. IEEE, 2008.

[14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[15] Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically generating inputs of death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):10, 2008.

[16] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. Detecting and escaping infinite loops with jolt. ECOOP'11, 2011.

[17] Antonio Carzaniga, Alessandra Gorla, Andrea Mattavelli, Nicolò Perino, and Mauro Pezzè. Automatic recovery from runtime failures. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 782–791.

[18] E. Ceesay, J. Zhou, M. Gertz, K. Levitt, and M. Bishop. Using type qualifiers to analyze untrusted integers and detecting security flaws in C programs. *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 1–16, 2006.

[19] R. Chinchani, A. Iyer, B. Jayaraman, and S. Upadhyaya. ARCHERR: Runtime environment driven program safety. *Computer Security–ESORICS 2004*, pages 385–406, 2004.

[20] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin C. Rinard. Inference and enforcement of data structure consistency specifications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 233–244, 2006.

[22] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 760–770. IEEE Press, 2012.

[23] Will Drewry and Tavis Ormandy. Flayer: Exposing application internals. In *Proceedings of the first USENIX workshop on Offensive Technologies*, pages 1–9. USENIX Association, 2007.

[24] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009.

[25] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[26] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[27] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: a guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX conference on Security*, pages 49–64. USENIX Association, 2013.

[28] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '12', pages 431–450. ACM, 2012.

[29] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. ICSE '12, 2012.

[30] Fan Long, Vijay Ganesh, Michael Carbin, Stelios Sidiroglou, and Martin Rinard. Automatic input rectification. MIT-CSAIL-TR-2011-044.

[31] Fan Long, Stelios Sidiroglou-Douskos, Deokhwan Kim, and Martin Rinard. Sound input filter generation for integer overflow errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14', pages 439–452, New York, NY, USA, 2014. ACM.

[32] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via error shepherding. In *Proceedings of the 35th ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '14'. ACM, 2014.

[33] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[34] David Molnar, Xue Cong Li, and David A Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th conference on USENIX security symposium*, pages 67–82. USENIX Association, 2009.

[35] Vijay Nagarajan, Dennis Jeffrey, and Rajiv Gupta. Self-recovery in server programs. ISMM '09', 2009.

[36] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. PLDI '07, 2007.

[37] Gene Novark, Emery D Berger, and Benjamin G Zorn. Exterminator: automatically correcting memory errors with high probability. *ACM SIGPLAN Notices*, 42(6):1–11, 2007.

[38] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. SOSP '09. ACM, 2009.

[39] Feng Qin, Joseph Tucek, Yuanyuan Zhou, and Jagadeesan Sundaresan. Rx: Treating bugs as allergies–a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 2007.

[40] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *OSDI*, pages 303–316, 2004.

[41] Martin C. Rinard. Acceptability-oriented computing. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2003, October 26-30, 2003, Anaheim, CA, USA*, pages 221–239, 2003.

[42] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, and Tudor Leu. A dynamic technique for eliminating buffer overflow vulnerabilities (and other memory errors). In *ACSAC*, pages 82–90, 2004.

[43] J Röning, M Lasko, A Takanen, and R Kaksonen. PROTOS — systematic approach to eliminate software vulnerabilities. *Invited presentation at Microsoft Research*, 2002.

[44] D. Sarkar, M. Jagannathan, J. Thiagarajan, and R. Venkatapathy. Flow-insensitive static analysis for detecting integer anomalies in programs. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 334–340. ACTA Press, 2007.

[45] R.C. Seacord. *The CERT C Secure Coding Standard*. Addison-Wesley Professional, 2008.

[46] Koushik Sen, Darko Marinov, and Gul Agha. *CUTE: A Concolic Unit Testing Engine for C*, volume 30. ACM, 2005.

[47] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[48] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. Assure: Automatic software self-healing using rescue points. In *ASPLOS*, pages 37–48, 2009.

[49] Stelios Sidiroglou, Eric Lahtinen, Nathan Rittenhouse, Paolo Piselli, Fan Long, Deokhwan Kim, and Martin Rinard. Automatic Integer Overflow Discovery Using Goal-Directed Conditional Branch Enforcement. In *ASPLOS*, 2015.

[50] Stelios Sidiroglou, Michael E Locasto, Stephen W Boyd, and Angelos D Keromytis. Building a reactive immune system for software services. In *Proceedings of the general track, 2005 USENIX annual technical conference: April 10-15, 2005, Anaheim, CA, USA*, pages 149–161. USENIX, 2005.

[51] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by multi-application code transfer. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 20015.

[52] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Pearson Education, 2007.

[53] Wang Tielei, Wei Tao, Lin Zhiqiang, and Zou Wei. IntScope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. In *16th Annual Network & Distributed System Security Symposium*, 2009.

[54] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of the 31st IEEE Symposium on Security & Privacy (Oakland'10)*, 2010.

[55] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M.F. Kaashoek. Improving integer security for systems with KINT. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 163–177. USENIX Association, 2012.

[56] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou. IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. *Computer Security–ESORICS 2010*, pages 71–86, 2010.