Diagnosis and Emergency Patch Generation for Integer Overflow Exploits

Tielei Wang, Chengyu Song, and Wenke Lee

School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA
{tielei,chengyu,wenke}@cc.gatech.edu

Abstract. Integer overflow has become a common cause of software vulnerabilities, and significantly threatens system availability and security. Yet protecting commodity software from attacks against unknown or unpatched integer overflow vulnerabilities remains unaddressed. This paper presents SoupInt, a system that can diagnose exploited integer overflow vulnerabilities from captured attack instances and then automatically generate patches to fix the vulnerabilities. Specifically, given an attack instance, SoupInt first diagnoses whether it exploits integer overflow vulnerabilities through a dynamic data flow analysis based mechanism. To fix the exploited integer overflows, SoupInt generates patches and deploys them at existing, relevant validation check points inside the program. By leveraging existing error-handlers for programmer-anticipated errors to deal with the unanticipated integer overflows, these patches enable the program to survive future attacks that exploit the same integer overflows. We have implemented a SoupInt prototype that directly works on x86 binaries. We evaluated SoupInt with various input formats and a number of real world integer overflow vulnerabilities in commodity software, including Adobe Reader, Adobe Flash Player, etc. The results show that SoupInt can accurately locate the exploited integer overflow vulnerabilities and generate patches in minutes.

1 Introduction

Zero-day attacks that exploit previously unknown software vulnerabilities are one of the most serious threats to cyber security. Once an exploit instance against commodity applications is captured in the wild [23, 27, 29], a pressing task for defenders is to diagnose the exploited vulnerabilities. Furthermore, since it usually takes a very long time for software vendors to release a patch [14], there continually exists a great demand for efficient and effective schemes to protect the vulnerable systems before the official vendor patches are available.

Particularly, in recent years, integer overflow vulnerabilities, one of the most serious software errors, are frequently discovered in widely used software and exploited by more and more real world attacks via malicious images, PDFs, Flash, and so forth [3]. Despite the considerable efforts made in the area of exploit diagnosis (e.g., [27, 29, 42, 45]) and prevention (e.g., [5, 7, 9–11]), most of them focus on memory corruption errors, and determining whether a wild-captured attack is exploiting integer overflow vulnerabilities and then preventing similar attacks remain unaddressed.

Solving these problems faces several challenges. First, diagnosis of integer overflow exploits needs a way to distinguish harmful integer overflows from benign ones.

S. Dietrich (Ed.): DIMVA 2014, LNCS 8550, pp. 255–275, 2014.

As shown in much existing work [4, 13, 38], benign or intentional integer overflows are very common in programs due to some general calculations such as hashing and random number generation. This means that an exploit instance can usually trigger a number of integer overflows during the execution of the program, making it very difficult to pinpoint the harmful one, if there is any.

Second, to prevent similar attacks that exploit the same vulnerabilities, one of the most popular defense mechanisms is to automatically generate patches to fix the vulnerabilities. Unfortunately, most existing patch generation systems such as [18,41] need source code of target programs, thus are not suitable for protecting COTS (Commercial off-the-shelf) programs. In addition, many researchers proposed to learn certain signatures from attack instances, and then further use the signatures to identify and discard malicious inputs. However, exploit-specific signatures can be easily evaded by obfuscation or polymorphic techniques [11, 26]; and vulnerability-specific signatures (such as [5, 7, 9–11]), despite being much more robust against polymorphic attacks, cannot handle encrypted or compressed inputs and may produce too many false negatives when input formats contain iterative fields or floating fields (see Section 2.1).

In this paper, we introduce SoupInt, a system designed to cooperate with existing exploit capture systems (e.g., [23,27]) to further identify the exploited integer overflow vulnerabilities in x86 binaries and generate emergency patches. As a temporary protection scheme, the generated patches can protect the vulnerable programs from similar attacks against the same vulnerabilities until official vendor patches are available.

```
310 HGLOBAL WinSalBitmap::ImplCreateDIB( const Size& rSize, USHORT nBits, const BitmapPalette& rPal )
311 {
      ...
314   HGLOBAL hDIB = 0;
315
316   if(rSize.Width()&&rSize.Height())//relevant validation checks; we deploy a patch here to avoid the overflow
317   {
318      const ULONG nImageSize = AlignedWidth4Bytes(nBits*rSize.Width())*rSize.Height(); //integer overflow
319      const USHORT nColors = ( nBits <= 8 ) ? ( 1 << nBits ) : 0;
320
321      hDIB = GlobalAlloc( GHND, sizeof( BITMAPINFOHEADER ) + nColors * sizeof( RGBQUAD ) + nImageSize );
      ...
350   }
351
352   return hDIB;
```

Fig. 1. Integer Overflow Vulnerability (CVE-2012-1149) in OpenOffice.org 3.3.0

Specifically, given an exploit instance captured by existing exploit detection systems (e.g., [23, 27, 29]), SoupInt first runs the vulnerable program with this exploit, and catches all integer overflows at runtime through binary instrumentation. To solve the challenge of distinguishing harmful integer overflows from benign ones, SoupInt leverages dynamic data flow analysis [16] to track the propagation of the overflows. If SoupInt finds that an integer overflow affects security sensitive operations (e.g., affecting the size parameters of memory allocation functions), SoupInt determines this integer overflow is harmful and this attack instance is exploiting an integer overflow vulnerability.

Next, inspired by the concept of error virtualization proposed in [33, 34], we design a novel method to automatically generate emergency patches for the exploited integer

overflow vulnerability. Our key observation is that programs usually perform some validation checks on input data and are able to correctly handle certain anticipated invalid inputs. Although such validation checks may be irrelevant or insufficient to prevent integer overflows, we can generate a patch and deploy it at such validation points so that the patched program can detect the integer overflow and make use of existing error handling code to survive attacks. We call this technique *local error virtualization*.

To better demonstrate our idea, take a real integer overflow vulnerability found in OpenOffice.org (Figure 1) as an example. The integer overflow vulnerability is in line 318 and can cause an undersized memory allocation at line 321, which eventually results in a heap overflow. Prior to the vulnerability point, the function checks if either `rSize.Width()` or `rSize.Height()` is zero. If so, it will directly return a `NULL` pointer, which can be correctly handled by the callers.

To fix this vulnerability, SoupInt will generate a patch and deploy it in line 316. The patch is able to test whether a concrete execution context will trigger the integer overflow in line 318, in which case, the patch will redirect the control flow to line 352, and return a `NULL` pointer, avoiding the integer overflow and surviving the attack by using internal existing error handler. Note, although this example is at source code level, SoupInt directly works on x86 binary executables.

To verify whether this idea is widely applicable, we manually investigated all CVE entries for publicly known integer overflow vulnerabilities in the Linux kernel (from 2009 to April 2012), the GNU C Library, and the GNU Image Manipulation Program (GIMP), and the corresponding patches. The result shows that for 84.9% (i.e., 26 of 32 CVE entries) of the integer overflow vulnerabilities the programs have incomplete validation checks on variables involved in the vulnerabilities, and patches can usually be deployed at these existing validation points.

Although this idea of local error virtualization is very intuitive, our implementation needs to address two technical challenges. First, SoupInt needs to choose proper patch deployment points for a given integer overflow vulnerability. To solve this challenge, SoupInt records the execution trace of the vulnerable program on the attack instance. It then employs a backward-forward slicing algorithm to identify the checks on the variables that are relevant to the harmful integer overflow operation, i.e., *relevant checks*. Finally, SoupInt uses heuristics derived from our manual analysis to select *validation checks* from these relevant checks as the patch deployment points.

The second challenge is, given a candidate patch deployment point, SoupInt needs to generate a patch that should be able to predict whether the integer overflow will be triggered by the execution context at the deployment point. To do this, SoupInt employs dynamic symbolic execution to calculate a symbolic predicate that represents the integer overflow condition and collect related trace constraints. At runtime, the patch will check whether these predicates are satisfiable for a concrete execution context. For malicious inputs that make such symbolic predicates satisfiable, the patch will alter the program's control flow to existing error handling code, and essentially transfers the unanticipated integer overflow errors to an anticipated error. Our patch generation scheme is significantly different from vulnerability signature generation systems (e.g., [5, 6, 12]) because SoupInt deploys the patch inside the programs. This new design makes SoupInt

effective even when the input data is encrypted or compressed, which is hardly handled by existing vulnerability signature systems.

In summary, this paper makes the following contributions:

- We developed a dynamic dataflow tracking based mechanism to accurately diagnose the exploited integer overflow vulnerabilities from wild-captured attack instances.
- We designed a novel approach named local error virtualization to fix integer overflow vulnerabilities by automatically generating and deploying patches at existing relevant validation check points. Unlike vulnerability signatures, our patches, which are much closer to manual patches, can enhance existing validation checks and block malicious inputs based on existing error handling functionalities.
- We have implemented a prototype system called SoupInt for x86 binaries. We apply SoupInt to ten real-world integer overflows in widely used commodity applications including Adobe Reader, Adobe Flash Player, Apple QuickTime, and Yahoo Messenger, and test ten different input formats. The results show that SoupInt can locate harmful integer overflows and quickly generate patches ***in minutes***, without relying on input specification or source code. Our patches can identify exploits ***in milliseconds*** without false positives, and enable programs to survive successfully.

The rest of the paper is organized as follows. Section 2 compares our research to related work. Section 3 describes the design of SoupInt algorithms and system components. Section 4 presents the implementation and evaluation of SoupInt. Section 5 discusses limitations and future work and Section 6 concludes the paper.

2 Related Work

2.1 Input Filter and Vulnerability Signature

A general solution to protect programs from attacks against unpatched vulnerabilities is to filter the malicious inputs based on exploit signatures or vulnerability signatures. A considerable number of techniques have been developed to generate such signatures [11, 20, 26]. However, they heavily rely on either knowledge of the input formats or specific features of the exploits.

A more robust way is to generate vulnerability-specific signatures that may be able to detect all attacks exploiting the same vulnerability [5, 7, 10, 11, 25, 36]. To automatically generate such signatures, many systems such as [5, 9, 10] take the original data in a captured exploit as symbolic values, and employ symbolic execution to extract trace conditions. The collected constraints and the vulnerability trigger condition are the vulnerability signature.

However, existing vulnerability signature generation systems have two major limitations. First, it is very hard for them to generate a signature based on symbolic execution if the input data is encrypted, obfuscated, or compressed [25]. It is a very practical issue since encryption and compression have been widely used (such as the HTTPS protocol and the Open Document format).

Second, these systems may have high false negatives if the vulnerability is triggered by the input values that do not have fixed offsets in the input format. For example,

many integer overflow vulnerabilities in JPEG File Interchange Format (JFIF)[1] parsers are caused by the `width` and the `height` fields of the images. However, instead of storing the `width` and `height` fields at fixed offsets, the JPEG format uses a special byte sequence to annotate these two fields. Thus, to locate the two fields, the parsers have to iteratively identify the byte sequence first. This process will introduce a number of unnecessary conditions. Given the trace executed by the exploit sample, the signatures generated by these systems can only detect the exploits that store `width` and `height` at the same offsets as the exploit sample. This is one reason why Cui et al. [12] estimate that Vigilante [10] would be effective for only 6 of the 25 vulnerabilities selected from Microsoft Security Bulletins.

In this paper, we also use symbolic execution to generate patches. However, our goal is not to simply filter malicious inputs before they are passed to the vulnerable program, but to fix the vulnerability by enhancing existing input validation checks inside the program. A key technical difference between our work and existing vulnerability signature generation systems is that we treat certain internal variables of the program as symbolic values, instead of original concrete input data. This makes our technique much less sensitive to the input formats. In addition, the patches are deployed inside the vulnerable program, and can be effective even when input data is encrypted or compressed.

2.2 Integer Overflow Detection and Prevention

Many approaches have been proposed to prevent integer overflows at the source code level [4, 8, 13, 39, 44]. However, these approaches usually have to kill the program once an integer overflow happens at runtime, which essentially transfers the integer overflow issues to denial-of-service attacks. Our patch can avoid the harmful integer overflows and employ existing error handling code to survive the attacks. Furthermore, compared with previous work on integer overflow vulnerability detection tools such as [24, 38], our work offers a way to automatically fix the detected vulnerabilities.

There has been much previous work on binary program static analysis and type inference [17, 19]. Our work could leverage these approaches to further recover the type information and reduce redundant integer overflow checks. We use dynamic data flow analysis technique [27] to track the propagation of integer overflows. Recently, many researchers propose various optimization methods for dynamic dataflow analysis such as [15, 16, 31], which can also be integrated into SoupInt to improve the performance.

2.3 Attack Diagnosis and Error Recovery

A number of diagnosis techniques (such as [27, 29, 42, 45]) have been proposed to automatically analyze an attack process, usually with an emphasis on illustrating how the program counter is controlled. In comparison, our paper proposes an approach to diagnose whether an attack is specifically exploiting integer overflow vulnerabilities.

ClearView [28] is designed to automatically patch errors in deployed software programs by enforcing the invariants that are learned from normal executions. However, ClearView is limited by what kinds of invariants it can learn, and may miss the root

[1] `http://www.w3.org/Graphics/JPEG/jfif3.pdf`

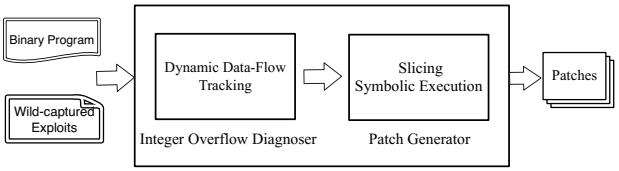

Fig. 2. The SoupInt Architecture

causes of a vulnerability. According to its evaluation results, ClearView fails to generate a patch for the heap overflow vulnerability in Firefox, which is actually caused by an integer overflow.

Sidiroglou et al. [34] introduce a nice concept of error virtualization, and further improve the idea and propose **rescue points** in ASSURE system [32,33]. A rescue point is a program location where the program checks return values from certain functions and dispatches programmer-anticipated errors to corresponding handlers. Essentially, rescue points are the *validation checks on function return values*. In our work, we generalize the idea by identifying *validation checks on the variables* that are involved in integer overflow vulnerabilities, and generate patches to enhance such validation checks.

ASSURE [33] and other similar systems such as [30, 35] rely on checkpoint-replay mechanism that can recover the execution after a fault *really* happens. However, continuous attacks against the same vulnerability will cause a significant number of expensive recovery efforts and may result in a denial-of-service. Our work does not have this limitation because our mechanism can generate patches and eliminate integer overflow vulnerabilities.

3 System Design

SoupInt takes a vulnerable program and a wild-captured exploit as inputs, diagnoses whether integer overflow vulnerabilities are exploited, and then generates emergency patches to fix them. Figure 2 shows the architecture of SoupInt. Note that we position SoupInt as an offline analysis system and assume that the exploit instances have been captured by existing detection systems (e.g., [23, 27, 29]).

The rest of this section is organized as follows. Section 3.1 introduces the integer overflow vulnerability diagnoser, which is responsible for capturing integer overflows at runtime and identifying harmful integer overflows. Section 3.2 describes the patch generator, which is used to select patch deployment points and generate patches.

3.1 Integer Overflow Vulnerability Diagnoser

This component has two goals: (1) it diagnoses whether a given attack instance exploits an integer overflow vulnerability or not; and (2) if so, it accurately locates where the harmful integer overflow happens. To achieve these goals, SoupInt first instruments all x86 instructions that may produce an integer overflow to detect overflows occurred during runtime. For the integer overflows that can be detected through the status register

(i.e., EFLAGS), SoupInt directly checks if a certain flag is set after that instruction is executed. For example, for signed ADD, SoupInt checks whether the overflow flag OF is set; and for unsigned ADD, SoupInt checks the carry flag CF.

For the integer overflows that cannot be detected through the status register, SoupInt pre-calculates the result before the instruction is executed and checks whether the result overflows. For example, the LEA instruction, designed to compute effective addresses, is widely used as an arithmetic operation. This instruction computes an expression of the form "base+index*scale+offset" and does not affect EFLAGS. For this instruction, SoupInt checks if each sub-expression overflows.

The challenge here is that binary programs do not preserve type information (i.e., signed or unsigned). To recover this information, we built a simple type inference tool based on previous work [38], which retrieves partial type information from signed/unsigned comparisons and arguments to known library/system APIs and propagates it based on classic data flow analysis. For instructions whose type information remains unknown after the static type inference, SoupInt performs both signed and unsigned overflow checks.

Once an integer overflow is detected at runtime, SoupInt then employs dynamic data flow analysis [16] to diagnose whether this integer overflow is harmful or not. Specifically, SoupInt assigns the overflow value a unique tag (i.e., the address of the instruction) and tracks the propagation of this tag according to dynamic data flow dependence. If SoupInt finds that a tagged value is used in security sensitive operations, it considers this integer overflow as harmful. Since allocating a buffer of incorrect size is the most typical result of integer overflow vulnerabilities [44], SoupInt currently treats the size parameters of memory allocation functions (such as malloc, calloc, HeapAlloc, and VirtualAlloc) and the size parameters of memory manipulation functions (such as memset, memcpy and memmove) as sensitive sinks. In the future, we can also add more sinks like loop bound checks and array index calculation.

3.2 Patch Generator

After identifying an integer overflow vulnerability, SoupInt re-runs the vulnerable program with the attack instance, and records a detailed execution trace, which contains accessed memory addresses and values, and accessed registers and their values of each instruction. Next, SoupInt offline analyzes the execution trace to identify candidate patch deployment points on the execution trace and then generates a corresponding patch using different policies. Finally, SoupInt tests whether the patches can fix the integer overflow vulnerability without breaking the program's normal execution.

Patch Deployment Point Discovery. A patch deployment point is a relevant validation check point. We start by introducing these terminologies and then describe the discovery algorithms. For x86 binaries, a conditional check (i.e., the conditional statement if in C/C++ programs) consists two instructions: an instruction C that affects the flag register and a conditional jump instruction J that depends on the result of C. So we use the pair (C, J) to indicate a check. Furthermore, let O be the integer overflow instruction.

Given an instruction i, we use $DataSlice(i)$ to represent the set of instructions on the trace that affect the values used in the instruction i through data flow dependencies.

This is different from traditional dynamic slicing [1] that considers both data flow and control flow dependencies.

Relevant Checks. A check (C, J) is relevant to an integer overflow instruction O if it tests a variable that has some relationships with the integer overflow. More specifically, if $DataSlice(C) \cap DataSlice(O) \neq \emptyset$, then (C, J) is relevant to O. For example, consider the following code:

```
1.x = input();
2.y = x;
3.z = x;
4.if(z==0){//relevant check to line 6, although z has no data flow dependence on y.
5.   handle_error(); return;}
6.y = y * 256; //harmful integer overflow
...
```

Assume that SoupInt has found the integer overflow vulnerability in line 6 and has recorded a trace [1, 2, 3, 4, 6]. Then $DataSlice(6)$ is $\{1, 2\}$ and $DataSlice(4)$ is $\{1, 3\}$. Since they have line 1 in common, the check statement at line 4 is relevant to the integer overflow at line 6.

Validation Checks. A check is a validation check in this paper if it is designed to identify the programmer-anticipated invalid values.

Identify Relevant Checks. We consider two types of relevant checks: 1) those located *before* the integer overflow instruction, and 2) those located *between* the integer overflow instruction and the sensitive operation where the overflowed value is used. To

```
   Input: Trace: Execution Trace, O: Integer Overflow Point
   Output: relevant checks
1  liveVars ← O.use(); //The inputs of O
2  DataSlice ← [];
3  tempVars ← [];
4  foreach inst from O to Trace[0] do
5      if liveVars==∅ then
6          break;
7      if inst.define() ∩ liveVars ≠ ∅ then
8          DataSlice.push(inst);
9          tempVars.push(liveVars ∩ inst.define());
10         liveVars ← liveVars - inst.define();
11         liveVars ← liveVars ∪ inst.use();
12 liveVars ← ∅;
13 forwardSlice ← [];
14 foreach inst from Trace[0] to O do
15     if inst in DataSlice then
16         liveVars ← liveVars ∪ tempVars.pop();
17         forwardSlice.push(inst);
18         continue;
19     if inst.use() ∩ liveVars ≠ ∅ then
20         forwardSlice.push(inst);
21         liveVars ← liveVars ∪ inst.define();
22     else
23         liveVars ← liveVars - inst.define();
24     if inst.isConditionalJump() then
25         recordRelevanCheck(inst);
```

Fig. 3. Backward-forward Slicing Algorithm

identify relevant checks *before* the integer overflow instruction, SoupInt uses the algorithm in Figure 3. The algorithm takes an execution trace T and an integer overflow instruction O as inputs, and mainly consists of two loops.

Table 1. Backward-forward Slicing Example

Trace	$inst.def()$	$inst.use()$	liveVars	tempVars	liveVars	tempVars
			Backward slicing from 7 to 1		Forward slicing from 1 to 7	
1. x=GetInt();	{x}	-	{q}	{x}	{x}	{x}
2. y = x;	{y}	{x}	{x,q}	{y}	{x,y}	{y}
3. z = x;	{z}	{x}	{y,q}	-	{x,y,z}	-
4. if(z==0)	-	{z}	{y,q}	-	{x,y,z}	-
5. s=y, p=q;	{s,p}	{y,q}	{y,q}	{s}	{x,y,z,s}	{s}
6. p=use(p);	{p}	{p}	{s}	-	{x,y,z,s}	-
7. malloc(s*4);	-	{s}	{s}	{}	{x,y,z,s}	{}
			DataSlice: {7,5,2,1}		*forwardSlice:* {1,2,3,4,5,7}	

We use the example trace (showing C source code for clarity purposes) in Table 1 to illustrate the algorithm. The first loop of the algorithm is designed to compute $DataSlice(O)$ by backwards traversing the define-use chain in the trace. $Inst$.define() and $inst$.use() represent the variables (i.e., registers and memory addresses) that are defined and used by $inst$ respectively. A particular problem is, an instruction usually defines multiple variables, but only some of them are relevant to the overflowed integer operation. So naively tracking all variables in define-use chain will cause some irrelevant checks to be added into the set of relevant checks by the second loop. To solve this problem, for each instruction $inst$ in $DataSlice(O)$, SoupInt uses $tempVars$ to only record the variables that can affect the overflowed instruction O (line 9 in Figure 3). For example, the "`push eax`" instruction modifies both the stack pointer and the memory location on the top of the stack, so its $Inst$.define() includes both the `esp` register and the memory `[esp]`. But if only the value stored in the memory address (`[esp]`) can affect O, then only this address will be recorded in $tempVars$. For another example, the line 5 in Table 1 defines both `s` and `"p"`. Since the definition of `s` causes the line to be sliced into $DataSlice$, $tempVars$ records `"{s}"` for line 5.

The second loop is designed to generate a forward slice. More specifically, the loop takes the instructions in $DataSlice(O)$ as slicing criteria, and only adds the corresponding element in $tempVars$ to $liveVars$. This means only variables that can affect the integer overflow instruction O are added to $liveVars$. Essentially, the forward slice tracks the propagations of the variables that can affect O. Since the flag register is considered in the define-use chain, the conditional jump instructions in the forward slice must have directly or indirectly data dependency on the variables that can affect O. Therefore, the checks in the forward slice are relevant checks to the integer overflow instruction. For example, line 4 checks variable `z`; although `z` has no data flow dependence on `s` that causes an integer overflow in line 7, our forward slice contains line 4. Furthermore, at line 5, although `p` is also in $inst$.def(), because of the presence of $tempVars$, only `s` is added into $liveVars$, which avoids line 6 from being added into $forwardSlice$.

To find relevant checks *between* the integer overflow instruction and the exploit point, we can simply extend the second loop in Figure 3 to slice from the first instruction in the trace to the point where the overflowed value is used in the security sensitive operation.

Identify Validation Checks. Since not all relevant checks are validation checks, SoupInt further refines the result according to the following heuristics:

Heuristics I. A validation check usually compares a variable with a constant value, such as checking whether a variable is zero or greater than a constant boundary value.

Heuristics II. Following the branch for invalid inputs of a validation check, the function is most likely to return quickly. We use three basic blocks as the threshold.

Heuristics III. If a validation check and the integer overflow point are in the same function, the integer overflow point is usually control dependent on the validation check. In other words, whether the integer overflow point will be executed is determined by the result of the validation check.

Only the checks that satisfy all of the three heuristics are selected as validation checks. These heuristics are based on our manual inspection of the 32 real-world vulnerabilities in the Linux kernel (from 2009 to April 2012), the GNU C Library, and the GNU Image Manipulation Program (GIMP), and represent the most common cases.

Patch Generation. After identifying candidate patch deployment points, SoupInt continues to generate a set of candidate patches. According to the position of the patch point, SoupInt has three types of patch generation policy.

Policy I. If the control flow reaches the candidate patch point *before* the integer overflow happens, SoupInt employs dynamic symbolic execution to generate a patch that can forestall the integer overflow by changing the control flow to the branch for handling invalid values.

Specifically, SoupInt performs forward dynamic symbolic execution from a candidate patch point to the integer overflow point along the recorded trace. Since dynamic symbolic execution has been presented in much literature, we do not elaborate it here. A key challenge in our scenario is how to choose the initial symbolic values. If we treat all the registers and the whole memory as symbolic values, we could collect all the symbolic trace constraints; but the result will contain too many unnecessary constraints and lead to high false negative rate. Therefore, SoupInt only takes the variables that can affect the values used in the integer overflow operation as symbolic values. The slicing algorithm in Figure 3 can already be used to identify such variables (i.e., the variables in $liveVars$). Based on these initial symbolic variables, SoupInt symbolically executes the instructions that access symbolic values, and concretely executes the instructions that do not access symbolic values.

When the dynamic symbolic execution stops at the integer overflow operation, SoupInt generates a set of symbolic predicates that describes (1) how the symbolic values are used in the integer overflow operation and; (2) path constraints for the execution to reach the integer overflow point. By inserting a new symbolic predicate that represents the overflow condition, this set of predicates can be used to determine whether the integer overflow will be triggered during runtime. SoupInt then exports these symbolic predicates to a file in the SMT-LIB format [2].

For example, the left side of Figure 4 shows a snippet of an execution trace and the right side also presents the corresponding symbolic execution process. Assume we have detected an integer overflow vulnerability at line 4. In this case, SoupInt will select line 2 as the patch deployment point, assigns `eax` and `ebx` initial symbolic values, say `eax_0` and `ebx_0` respectively, and perform symbolic execution from this point. As a result, SoupInt computes that the size argument of `malloc` is `eax_0*(ebx_0+4)`, and generates a predicate for the multiplication overflow condition.

A patch of this type is a function that is deployed at corresponding validation check point. Every time the control flow reaches the validation check point (e.g., line 2 in Figure 4), this patch function will be invoked. Specifically, the patch function loads the symbolic predicate, instantiates the symbolic values according to the concrete execution context, and check its satisfiability. If the symbolic predicate is satisfiable, which means that the execution context will reach the integer overflow point and trigger an overflow, the patch function changes the program's control flow to the branch for invalid inputs.

In the above example, a patch function is deployed at line 2. When it is invoked, the patch function instantiates symbolic `eax_0` and `ebx_0` with the values of registers `eax` and `ebx`, respectively, and alters the program control flow to the `err` branch after it finds that the patch predicates in Figure 4 are satisfiable.

Policy II. If the control flow reaches the candidate patch point *after* the integer overflow happens, SoupInt generates a patch that can alter the control flow at the validation check point before the control flow reaches the exploit point, if it captures the integer overflow at runtime. Specifically, the patch consists of three components. The first component uses the Thread-Local Storage (TLS) method to allocate a global alarm flag for each thread. The second component, which is deployed at the integer overflow point, is responsible for setting the global alarm flag if the integer overflow happens at runtime. The third component, which is deployed at the validation check point, alters the program's control flow to the branch for invalid inputs if the alarm flag is set, and then resets the alarm flag. Figure 5 shows a high level example. The code in boxes represents the corresponding patch components.

	New Symbolic Map
1. test eax, eax	eax=eax_0 ebx=ebx_0
2. jz err	
3. add ebx,4	ebx=ebx_0+4 eax_0!=0
4. imul eax, ebx	eax=eax_0*(ebx_0+4) eax_0!=0
5. push eax	
6. call malloc	Patch Predicates: "eax_0!=0 && (eax_0*(ebx_0+4)) overflows"

Fig. 4. Symbolic Execution Example

```
1. Init_TLS_Flag();
size = count*4;
2. Set_TLS_Flag_If_Overflow();
if(size==0)
3. || if (TLS_Flag_Is_True())
        return err;
p = malloc(size);
return p;
```

Fig. 5. Policy II Example

Policy III. If SoupInt does not find any proper patch deployment points (e.g., no any validation check in the program), it generates a patch that performs a controlled exit if harmful integer overflow happens. There is a special case. If SoupInt finds the overflowed value affects a memory allocation and the program has memory failure checks,

SoupInt can generate a patch that forces the memory allocation function to return a NULL pointer if the integer overflow really happens.

Patch Test and Deployment. The generated patches can then be applied in various ways, such as static binary rewriting, dynamic binary rewriting or dynamic binary translation (e.g., PIN [21]). Since the details of these techniques are orthogonal to the topic of this paper, we will not discuss them here. In our current prototype system, we deploy patches using the PIN [21] platform, because PIN can attach to a running process without restarting it. Our PIN plugin loads the patch files and dynamically hooks the corresponding instructions according to the patch policy. It employs PIN's context manipulation APIs to manipulate a program's control flow, and uses PIN's Thread APIs to implement the thread local storage.

Before the final deployment, SoupInt will test the candidate patches using the standard patch testing procedure. First, we run the patched program with exploit samples to test whether a patch can prevent exploits against the integer overflow vulnerability. In practice, since we may only capture one or a few exploit samples, we can generate more malicious inputs by using the fuzzing technique, i.e., randomly modify the captured exploits. Many systems such as [12,40] use a similar approach to construct potential attack variants. A patch is considered effective if 1) the patched program survives all attack variants, and 2) the integer overflow does not happen at runtime for programs patched by Policy I and the integer overflow does not flow into security sensitive operations for the programs patched by Policy II and Policy III.

Second, we perform a regression test for each patch with normal inputs to check whether it affects the normal operations. If the patched program does not crash, fail, or generates different behaviors (compared with the original program with the same inputs), the patch is considered as useful and is ready for deployment.

4 System Evaluation

4.1 Implementation

We have implemented a prototype of SoupInt. Specifically, the integer overflow detector and tracker are implemented as plugins for PIN binary instrumentation platform (v2.11) [21]. We built a simple type inference tool based on our previous work [38] and extended our previous symbolic execution system [37] to generate symbolic predicates.

4.2 Experiment Setup

We evaluated SoupInt on its effectiveness and efficiency with ten integer overflow vulnerabilities in widely used applications, which involve ten kinds of input formats. Table 2 shows the basic information of these vulnerabilities, including the names, versions, availability of source code of the applications, the CVE identifiers for the vulnerabilities, and the corresponding input formats.

We chose these vulnerabilities according to the following steps. First, we only select the vulnerabilities in widely used programs. Specifically, we search for integer overflow

Table 2. Real World Integer Overflow Vulnerabilities

Software	Description	Version	Open Source	CVE ID	Input
Openoffice.org	Office productivity software suite	3.3.20	Y	CVE-2012-1149	ODT
VLC	Multimedia player	1.1.0	Y	CVE-2011-2194	XSPF
Yahoo Messenger	Instant messaging	11.5.0.152	N	CVE-2012-0268	JPEG
ACDSee	Image viewer	14.1	N	CVE-2012-1197	BMP
Opera	Web browser	11.6	N	CVE-2012-1003	HTML
Adobe Flash Player	Web browser plug-in	10.0.42.34	N	CVE-2010-2170	SWF
Adobe Reader	PDF viewer	9.1.3	N	CVE-2009-3459	PDF
RealPlayer	Multimedia player	SP 1.1	N	CVE-2010-3000	FLV
QuickTime Player	Multimedia Player	7.1.3	N	CVE-2007-0714	MPEG-4
Microsoft Linker	Key component of Microsoft Visual Studio	10.00.30319.01	N	N/A	PE
Summary: 10 integer overflows and 10 different input formats.					

vulnerabilities in the National Vulnerability Database[1] and the Secunia Vulnerability Database[2], and only select the integer overflows discovered in the widely used programs on the Windows x86 platform. Second, we further select the vulnerabilities whose exploits are available. Although we found a number of integer overflow vulnerabilities in Step 1, only few of them have publicly available exploits. To obtain the exploit samples, we contacted many discoverers of the vulnerabilities and also searched exploits in the Exploit-DB website. Finally, we chose the first 10 vulnerabilities that have exploits available.

We start by briefly introducing each vulnerability. Then, we present the effectiveness of SoupInt system in Section 4.3. We ran these applications with exploits and test whether SoupInt is able to locate the exploited integer overflow vulnerabilities and generate patches for these vulnerabilities. In Section 4.4, we present the efficiency of SoupInt system, including performance measurements of each component and the generated patches. All experiments ran on a Windows 7 virtual machine with 4GB of memory using VMware.

1. OpenOffice.org has an integer overflow vulnerability (as shown in Figure 1) when parsing JPEG objects embedded in a document in the Open Document (odt) format. The vulnerability can be triggered by overly large image dimensions of a JPEG object, and eventually results in a heap-based buffer overflow.
2. VLC player has an integer overflow in the XSPF playlist parser. The XSPF is in the XML format. An overly large value in the tag `<vlc:id></vlc:id>` can trigger the integer overflow and finally causes a heap overflow.
3. Yahoo Messenger has an integer overflow vulnerability. Malicious JPEG images with specially crafted image dimension values and color depth can trigger the integer overflow and eventually lead to a heap-based buffer overflow.
4. ACDSee, a popular image viewer, has an integer overflow vulnerability in the BMP image parser, which is caused by malformed dimension values of BMP images. The vulnerability can cause a heap-based buffer overflow.
5. The Opera web browser has an integer overflow vulnerability when calculating the buffer size for number arrays. Malicious JavaScript code can exploit the vulnerability by using a large integer argument to the typed array construction functions,

[1] `http://nvd.nist.gov/`
[2] `http://secunia.com/`

such as `Int32Array`, `Float32Array`, and `Int16Array`. The integer overflow eventually leads to a heap-based buffer overflow.

6. QuickTime has an integer overflow vulnerability that is caused by the size fields of the `udta` atoms within multimedia files in the MPEG-4 format. This vulnerability is able to trigger a heap based buffer overflow.
7. RealPlayer has an integer overflow vulnerability when parsing an FLV file with malformed `AMF` data, which can lead to a heap-based buffer overflow.
8. Adobe Reader has an integer overflow vulnerability in the FlateDecode stream parser, caused by the `/ParamX` parameter of a FlateDecode stream. This integer overflow can cause a heap overflow and finally leads to remote code execution. Note that this vulnerability was actively exploited in the wild in limited targeted attacks.
9. Adobe Flash Player has an integer overflow vulnerability when parsing embedded image data within SWF files. A crafted DefineBits tag within a SWF, which contains image data with malformed dimension values in the JPEG format, can trigger the vulnerability, and cause a heap-based buffer overflow.
10. Microsoft Linker, a key component of the Microsoft Visual Studio integrated development environment that links Common Object File Format (COFF) object files and libraries, has an integer overflow vulnerability when parsing PE files. The vulnerability is caused by the `NumberOfSymbols` field in the COFF file header within a PE (.exe) file, and leads to a heap-based buffer overflow.

Table 3. Attack Diagnosis Results

Software	Integer Overflow Vulnerability	Module	Offset	# Overflow Sites
Openoffice.org	`imul edi, edx`	vclmi.dll	0x1ad49f	1122
VLC	`lea esi, ptr [ecx*4+0x4]`	libplaylist_plugin.dll	0xfcd9	423
Yahoo Messenger	`imul eax, ebx`	YImage.dll	0x21531	354
ACDSee	`imul ebp, ecx`	IDE_ACDStd.apl	0x59639	288
Opera	`imul eax, dword ptr [esp+0xc]`	opera.dll	0x889f5b	428
Adobe Flash Player	`imul eax, ecx`	Flash10d.ocx	0x9165e	860
Adobe Reader	`lea edx, ptr [ecx*4+0x48]`	AcroRd32.dll	0xa60a5	1082
RealPlayer	`imul ecx, ecx, 0x23`	flvff.dll	0x8bc4	381
QuickTime Player	`add ecx, edi`	QuickTime.qts	0x295a74	567
Microsoft Linker	`lea edi, ptr [eax+eax*8]`	linker.exe	0xa2c10	88

4.3 Effectiveness

We ran the unpatched versions of applications in Table 2 with exploit samples, and used SoupInt to monitor the execution. SoupInt accurately locates the exploited integer overflow vulnerabilities, that is, SoupInt is able to capture the integer overflows at runtime, and then detects the overflow values flow into memory allocation functions. Table 3 summarizes the results. The second column presents the specific instructions where integer overflow happens, and the third and fourth columns show the corresponding modules and offsets. The last column reports the number of unique integer overflow sites. Note that, SoupInt detects **a large number** of integer overflows at runtime (the last column in Table 3), including both benign and harmful integer overflows, but only

the harmful ones (second column in Table 3) affect the memory allocations and need to be patched. These programs use different functions for allocations memory. For example, ACDSee and OpenOffice use `GlobalAlloc`, Yahoo Messenger uses `new()` operator, Adobe Flash player uses `malloc`,VLC uses `realloc`, and Microsoft Linker uses `RtlAllocateHeap`. By using dynamic data flow tracking, SoupInt is able to accurately locate the harmful integer overflows.

After locating the exploited integer overflow vulnerabilities, SoupInt continues to generate patches to fix these vulnerabilities. We manually verified that SoupInt correctly found the error handling branch by using the heuristics in Section 3.2. In summary, out of the 10 vulnerabilities in Table 2, SoupInt finds relevant validation checks before the integer overflows for 7, and successfully generates patches using Policy I; SoupInt does not find validation checks before the integer overflows, but finds validation checks after the integer overflows for 2, and generates patches using Policy II; SoupInt does not find any relevant validation checks for 1, and generates patches using Policy III.

Table 4. Policy I Patch Evaluation Results

Software	Relevant Checks	Validation Checks	# Final Patches
Openoffice.org	17	9	8
Yahoo Messenger	14	4	4
ACDSee	10	10	3
Opera	1	1	1
VLC	2	1	1
Adobe Reader	23	8	8
Microsoft Linker	1	1	1
Summary: successfully fixed these 7 vulnerabilities by using Policy I.			

Table 5. Policy II and III Patches

Policy Type	Software	Fixed
II	Adobe Flash Player	Y
	Quicktime	Y
III	RealPlayer	Y

Policy I. Table 4 shows Policy I evaluation results. The "Relevant Checks" column reports the number of relevant check points before the integer overflow points, identified by our slicing algorithm, and the "Validation Checks" column presents the number of candidate validation check points selected from the relevant checks. For each candidate relevant validation point, SoupInt generates a patch. Therefore, when a program checks inputs for multiple times at different places, SoupInt may generate multiple candidate patches for a single vulnerability. In this case, each of the candidate patches is evaluated independently. If a patch cannot prevent the program from crashing on malicious inputs or produces incorrect results on normal inputs, the patch cannot pass our tests. The "#Final Patches" column shows the number of successful patches that both survive the malicious inputs and enable the application to operate normally.

For OpenOffice, SoupInt finds 17 relevant checks and selects 9 of them as validation checks. We manually inspect the 9 validation check points in source code. We find (1) the function `get_sof` in the libjpeg package has two checks that test whether the image dimensions are signed less than or equal to zero; (2) the function `initial_setup` in the libjpeg package has two checks that test whether the image dimensions are signed greater than 65500; (3) a constructor function `Bitmap::Bitmap` of the Bitmap class has two checks on image dimensions to test whether they are zero and has one check on BitCount (i.e., the number of bits

per pixel of the image) to test whether it is greater than 8; and (4) the function `WinSalBitmap::ImplCreateDIB` as shown in Figure 1 has two checks at the line 316 that test whether the image dimensions are zero. SoupInt generate 8 successful patches (i.e., passing our tests) in the 9 patch points, except for the validation check on the BitCount in the constructor function of the Bitmap. While the original OpenOffice.org crashes when opening the crafted document file, the patched OpenOffice.org can successfully process the crafted document and provide normal functionalities such as editing the document and converting the document into other formats.

This Case Highlights the Advantage of our Approach. Since the input document file is in the Open Document format, which is a ZIP compressed archive, OpenOffice.org will first decompress the input file before parsing the malformed JPEG object. Due to the complicated decompression process, it is very difficult for the vulnerability signature generation systems such as [5, 9, 10] to generate a signature based on symbolic execution for this vulnerability.

For Yahoo Messenger, SoupInt finds 14 relevant checks, and further selects 4 of them as validation checks. The first two validation checks are used to test whether the dimension values of a JPEG image are signed less than or equal to zero, and the other two validation checks are used to test whether the dimension values are signed greater than 0xFFDC. SoupInt generates four patches and all of them are able to prevent the integer overflow.

For ACDSee, SoupInt finds 10 relevant checks. All of them are selected as validation checks. The interesting finding is that ACDSee does have integer overflow checks on the BMP image dimensions. However, these checks cannot prevent the integer overflow. Basically, ACDSee first promotes the signed 32-bit image dimensions to unsigned 64-bit integers, computes the multiplication result, and then uses a signed comparison to check whether the result is greater than 0x7fff. A correct check should use an unsigned comparison here. The malicious BMP image can pass the checks and trigger the integer overflow issue. This whole process contains multiple checks. SoupInt successfully generates 3 patches to fix the integer overflow issue.

For Opera web browser, SoupInt discovers one check before the integer overflow operation that tests whether the number of items in the array is zero. SoupInt further generates a patch and deploys the patch at the validation check point. **This case also highlights the advantage of our approach**. As malicious JavaScript code can easily use various obfuscation techniques, traditional vulnerability signature systems [5, 9, 10] are unlikely to identify and filter them without de-obfuscation. However, our patch is deployed inside the Opera browser and is able to resist all obfuscation techniques. In addition, the patch can also defeat the attacks via different JavaScript APIs, such as `Int32Array` and `Float32Array`.

For VLC player, SoupInt detects two relevant checks before the integer overflow, one of which is selected as the validation check. It tests whether the track ID (i.e., the value read from `<vlc:id>` element) is negative.

For Adobe Reader, among the 23 relevant checks, SoupInt identifies 8 validation checks. These validation checks are responsible for testing whether the input value read from the `/ParamX` parameter and corresponding intermediate variables are zero or negative. All of them are suitable for deploying patches.

For Microsoft Linker, SoupInt detects only one relevant check, which tests whether the `NumberOfSymbols` field is zero and is selected as a validation check. The patch deployed at this check point can successfully prevent the integer overflow.

Policy II & III. For the two vulnerabilities in Adobe Flash Player and QuickTime Player, although SoupInt does not find any relevant checks before the integer overflow operation, it detects the checks after the integer overflow and generates patches using Policy II. Interestingly, these checks after the integer overflow operations seem to be designed to detect the integer overflows, but they are insufficient. For example, the pseudocode for the vulnerability in Adobe Flash Player is shown as follows:

```
//w and h are the dimension values of a JPEG object
int tmp1 = w*4;
int size = tmp1*h; //integer overflow point
if(tmp1<=0 || h<=0 ||size<h || size<tmp1) //incorrect overflow checks
   goto _err;
ptr = malloc(size);
```

SoupInt generates patches that can alter the control flow to the error branch if the integer overflow occurs, essentially enhancing the existing overflow checks.

Table 6. System Performance Results

Software	Diagnosis(s)	Tracing(s)	Slicing(s)	Patching(s)
Yahoo Messenger	57	164	16	6.3
OpenOffice.org	181	210	53	10.2
ACDSee	123	206	18	8.8
Opera	105	332	49	6.5
VLC	112	134	28	8.6
Adobe Reader	99	361	71	21.5
Adobe Flash Player	144	344	52	N/A
QuickTime	78	217	73	N/A
RealPlayer	93	228	31	N/A
Microsoft Linker	37	66	27	12.1
Summary: diagnosing and patching were completed in minutes.				

Table 7. Patch Overhead

Software	Normal (μs)	Malicious (μs)
Yahoo Messenger	3190	4503
OpenOffice.org	5028	6572
ACDSee	1241	2442
Opera	727	761
Adobe Reader	597	1524
VLC	306	509
MS linker	1660	1819

For the vulnerability in RealPlayer, SoupInt does not find any validation checks. In fact, RealPlayer directly uses input data to calculate the size parameter of the `new` operator, without any sanity checks. Fortunately, SoupInt finds that RealPlayer has a check on the return value of the `new` operator. In this case, SoupInt generates a patch that can bypass the invocation to the `new` operator when the integer overflow happened and assign the EAX register (i.e., the return value) zero. The patch cannot stop the integer overflow, but avoids the heap overflow. The patched RealPlayer successfully survives the exploits.

Policy II and III results are summarized in Table 5. Our manual inspection shows that these patches in Table 5 were deployed at the post-dominators of the integer overflow operations (i.e., every path from the integer overflow operation to the exit of the function has to pass through our patch). If the integer overflow happens, the patch in Table 5 can prevent the overflown results from being used in security sensitive operations and are both complete and sound.

Overall, SoupInt is able to handle all the ten integer overflow vulnerabilities in Table 2, 7 of which are fixed by Policy I with symbolic predicate patches, 2 of which are fixed by Policy II, and 1 of which is fixed by Policy III.

4.4 Performance

We first report the performance of patch generation. Table 6 summarizes the evaluation results and presents the time spent on each primary step. In general, SoupInt can finish the attack diagnosis and patch generation in a few minutes. The second column shows the time spent on attack diagnosis, the "Tracing" and "Slicing" columns report the time spent on recording the execution traces and the time spent on slicing, and the last column shows the mean time spend on generating a symbolic predicate patch.

Next, we present the performance overhead caused by the patches per execution. For the patches generated from Policy I, the second and third column in Table 7 show the average execution time of the additional checks for normal and malicious inputs. For malicious inputs, the patches need to redirect applications' execution flows so it takes a bit more time. In all the cases, our patches are only executed once or a few times per execution and the overall overhead caused by the patches is completely negligible. The patches generated from Policy II or Policy III do not cause measurable performance overhead, compared to the case of running the programs in PIN [21] without any instrumentation.

5 Limitations and Future Work

In this section, we discuss the limitations of SoupInt and future work.

Scope. In our current implementation, SoupInt particularly handles the integer overflows that lead to incorrect memory allocations and movements, which are the most typical consequence of integer overflow vulnerabilities [39, 44]. While it is very easy to extend SoupInt to handle more integer overflow vulnerabilities that affect other sensitive functions, SoupInt does not handle integer overflows that do not have obvious sink points and lead to logic errors. Moreover, in the patch testing phase, we assume that a validation test suite of sufficient size is available.

Patch Overhead. Although the runtime overhead caused by our patches is trivial in our evaluation, it may be still unacceptable for performance-sensitive programs if the patches are deployed in the time critical parts. To alleviate the risk, we could optimize patch checks by translating the symbolic patches into simple predicates and improve the efficiency of patch checks.

Completeness and Soundness. In general, we cannot prove the completeness and soundness of the patches generated by SoupInt. It is well known that generating a complete and sound patch is very challenging, even for programmers [22, 43]. Since SoupInt generates symbolic predicate patches based on a single execution trace, our patches may have false negatives, i.e., the malicious inputs can trigger the integer overflow via a different program path and cannot be detected by our patches. In practice, we find that SoupInt can usually generate and deploy patches at control flow dominators or

post-dominators of the harmful integer overflow operations, in which case, the generated patches could be sound or complete. On the other hand, SoupInt now only treats the values that can affect the integer overflow operation as symbolic values. While this makes the patches more robust because a lot of unnecessary trace constraints are excluded, this may also cause false positives. For example, it is possible that our patches find an input will trigger the integer overflow, but in practice the input cannot reach the integer overflow operation. Note that in our evaluation, our patches do not generate false positives. The reason is that the overly large values detected by our patches have been able to indicate the whole input is invalid or malformed.

Future Work. In the future, we intend to extend SoupInt in two directions. First, we plan to improve the performance of the exploit diagnosis module so that SoupInt could be used as an online exploit detection tool. Second, we plan to extend SoupInt to fix other types of vulnerabilities such as buffer overflows and format string bugs using a similar idea of generating and deploying patches at existing validation check points.

6 Conclusion

In this paper, we presented SoupInt, a system that can automatically generate emergency patches from attacks against integer overflow vulnerabilities. SoupInt first uses the dynamic data flow analysis technique to diagnose the integer overflow vulnerabilities exploited by an attack instance, and then generates patches to eliminate these vulnerabilities using different policies. A key feature of SoupInt is that it deploys the patches at the existing relevant validation check points inside the vulnerable programs, and leverages the existing error handling code to deal with the unanticipated integer overflow vulnerabilities. Our experimental results on a number of real world integer overflow vulnerabilities in widely used commodity applications show that SoupInt can successfully locate harmful integer overflows and generate effective patches in minutes.

Acknowledgements. The authors would like to thank the anonymous reviewers for their valuable comments. This material is based upon work supported in part by the National Science Foundation under Grants No. CNS-1017265, CNS-0831300, and CNS-1149051, by the Office of Naval Research under Grant No. N000140911042, by the Department of Homeland Security under contract No. N66001-12-C-0133, and by the United States Air Force under Contract No. FA8650-10-C-7025. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Office of Naval Research, the Department of Homeland Security, or the United States Air Force.

References

1. Agrawal, H., Horgan, J.R.: Dynamic program slicing. SIGPLAN Not. 25, 246–256 (1990)
2. Barrett, C., Stump, A., Tinelli, C.: The smt-lib v2 language and tools: A tutorial (February 2011), `www.smtlib.org`

3. Bilge, L., Dumitras, T.: Before we knew it: an empirical study of zero-day attacks in the real world. In: CCS (2012)
4. Brumley, D., cker Chiueh, T., Johnson, R., Lin, H., Song, D.: Rich: Automatically protecting against integer-based vulnerabilities. In: NDSS (2007)
5. Brumley, D., Newsome, J., Song, D., Wang, H., Jha, S.: Towards automatic generation of vulnerability signatures. In: IEEE Symposium on Security and Privacy (May 2006)
6. Brumley, D., Wang, H., Jha, S., Song, D.: Creating vulnerability signatures using weakest preconditions. In: IEEE Computer Security Foundations Symposium (2007)
7. Caballero, J., Liang, Z., Poosankam, P., Song, D.: Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) RAID 2009. LNCS, vol. 5758, pp. 161–181. Springer, Heidelberg (2009)
8. Coker, Z., Hafiz, M.: Program transformations to fix c integers. In: ICSE (2013)
9. Costa, M., Castro, M., Zhou, L., Zhang, L., Peinado, M.: Bouncer: securing software by blocking bad input. In: ACM SIGOPS Symposium on Operating Systems Principles (2007)
10. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham, P.: Vigilante: end-to-end containment of internet worms. In: SOSP (2005)
11. Crandall, J.R., Su, Z., Wu, S.F., Chong, F.T.: On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In: CCS (2005)
12. Cui, W., Peinado, M., Wang, H.J., Locasto, M.E.: Shieldgen: Automatic data patch generation for unknown vulnerabilities with informed probing. In: IEEE Symposium on Security and Privacy (2007)
13. Dietz, W., Li, P., Regehr, J., Adve, V.: Understanding integer overflow in c/c++. In: ICSE (2012)
14. Frei, S., Tellenbach, B., Plattner, B.: 0-day patch - exposing vendors (in)security performance. In: BlackHat Europe (2008)
15. Jee, K., Portokalidis, G., Kemerlis, V.P., Ghosh, S., August, D.I., Keromytis, A.D.: A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In: NDSS (2012)
16. Kemerlis, V.P., Portokalidis, G., Jee, K., Keromytis, A.D.: libdft: practical dynamic data flow tracking for commodity systems. In: VEE (2012)
17. Lee, J., Avgerinos, T., Brumley, D.: Tie: Principled reverse engineering of types in binary programs. In: NDSS (2011)
18. Lin, Z., Jiang, X., Xu, D., Mao, B., Xie, L.: Autopag: Towards automated software patch generation with source code root cause identification and repair. In: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security (2007)
19. Lin, Z., Zhang, X., Xu, D.: Automatic reverse engineering of data structures from binary execution. In: NDSS (2010)
20. Long, F., Ganesh, V., Carbin, M., Sidiroglou, S., Rinard, M.: Automatic input rectification. In: ICSE (2012)
21. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: PLDI (2005)
22. Maurer, M., Brumley, D.: Tachyon: tandem execution for efficient live patch testing. In: USENIX Conference on Security Symposium (2012)
23. min Wang, Y., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.: Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In: Proceedings of the Network and Distributed Systems Security Symposium (2006)
24. Molnar, D., Li, X.C., Wagner, D.A.: Dynamic test generation to find integer bugs in x86 binary linux programs. In: Proceedings of the 18th USENIX Security Symposium (2009)

25. Newsome, J., Brumley, D., Song, D.: Vulnerability-specific execution filtering for exploit prevention on commodity software. In: NDSS (2008)
26. Newsome, J., Karp, B., Song, D.: Polygraph: Automatically generating signatures for polymorphic worms. In: IEEE Symposium on Security and Privacy (2005)
27. Newsome, J., Song, D.: Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In: NDSS (2005)
28. Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M.D., Rinard, M.: Automatically patching errors in deployed software. In: SOSP (2009)
29. Portokalidis, G., Slowinska, A., Bos, H.: Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (2006)
30. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: treating bugs as allergies—a safe method to survive software failures. In: SOSP (2005)
31. Ruwase, O., Chen, S., Gibbons, P.B., Mowry, T.C.: Decoupled lifeguards: enabling path optimizations for dynamic correctness checking tools. In: PLDI (2010)
32. Sidiroglou, S., Laadan, O., Keromytis, A.D., Nieh, J.: Using rescue points to navigate software recovery. In: IEEE Symposium on Security and Privacy (2007)
33. Sidiroglou, S., Laadan, O., Perez, C., Viennot, N., Nieh, J., Keromytis, A.D.: Assure: automatic software self-healing using rescue points. In: ASPLOS (2009)
34. Sidiroglou, S., Locasto, M.E., Boyd, S.W., Keromytis, A.D.: Building a reactive immune system for software services. In: USENIX Annual Technical Conference (2005)
35. Tucek, J., Newsome, J., Lu, S., Huang, C., Xanthos, S., Brumley, D., Zhou, Y., Song, D.: Sweeper: a lightweight end-to-end system for defending against fast worms. In: EuroSys (2007)
36. Wang, H.J., Guo, C., Simon, D.R., Zugenmaier, A.: Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In: Sigcomm (2004)
37. Wang, T., Wei, T., Gu, G., Zou, W.: Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. ACM Trans. Inf. Syst. Secur. 2 (September 2011)
38. Wang, T., Wei, T., Lin, Z., Zou, W.: IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In: NDSS (2009)
39. Wang, X., Chen, H., Jia, Z., Zeldovich, N., Kaashoek, M.F.: Improving integer security for systems with kint. In: OSDI (2012)
40. Wang, X., Li, Z., Xu, J., Reiter, M.K., Kil, C., Choi, J.Y.: Packet vaccine: black-box exploit detection and signature generation. In: CCS (2006)
41. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In: International Conference on Software Engineering (2009)
42. Xu, J., Ning, P., Kil, C., Zhai, Y., Bookholt, C.: Automatic diagnosis and response to memory corruption vulnerabilities. In: CCS (2005)
43. Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., Bairavasundaram, L.: How do fixes become bugs? – a comprehensive characteristic study on incorrect fixes in commercial and open source operating systems. In: FSE (2011)
44. Zhang, C., Wang, T., Wei, T., Chen, Y., Zou, W.: IntPatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 71–86. Springer, Heidelberg (2010)
45. Zhang, M., Prakash, A., Li, X., Liang, Z., Yin, H.: Identifying and Analyzing Pointer Misuses for Sophisticated Memory-corruption Exploit Diagnosis. In: NDSS (2012)