Xu *et al. Cybersecurity* (2020) 3:18
https://doi.org/10.1186/s42400-020-00058-2

Cybersecurity

RESEARCH Open Access

ELAID: detecting integer-Overflow-to-Buffer-Overflow vulnerabilities by light-weight and accurate static analysis

Lili Xu[1*], Mingjie Xu[1,2], Feng Li[1] and Wei Huo[1]

Abstract

The Integer-Overflow-to-Buffer-Overflow (IO2BO) vulnerability has been widely exploited by attackers to cause severe damages to computer systems. Automatically identifying this kind of vulnerability is critical for software security. Despite many works have been done to mitigate integer overflow, existing tools either report large number of false positives or introduce unacceptable time consumption. To address this problem, in this article we present a static analysis framework. It first constructs an inter-procedural call graph and utilizes taint analysis to accurately identify potential IO2BO vulnerabilities. Then it uses a light-weight method to further filter out false positives. Specifically, it generates constraints representing the conditions under which a potential IO2BO vulnerability can be triggered, and feeds the constraints to SMT solver to decide their satisfiability. We have implemented a prototype system ELAID based on LLVM, and evaluated it on 228 programs of the NIST's SAMATE Juliet test suite and 14 known IO2BO vulnerabilities in real world. The experiment results show that our system can effectively and efficiently detect all known IO2BO vulnerabilities.

Keywords: Integer-Overflow-to-Buffer-Overflow (IO2BO) vulnerability, Inter-procedural dataflow analysis, Taint analysis, Path satisfiability

Introduction

Integer overflow is one of the most common types of software vulnerabilities. According to the Common Vulnerability and Exploit (CVE) (Common Vulnerabilities and Exposures (CVE) 2020), integer overflow has become the second most critical type of coding errors, second only to buffer overflows (Christey and Martin 2007). If the malformed value generated by integer overflow (IO for short) is used for determining how much memory to allocate, it will cause a buffer overflow (BO for short), which is known as the Integer Overflow to Buffer Overflow vulnerability (CWE-680: IO2BO Vulnerabilities 2020). According to (Zhang et al. 2010), it is difficult to distinguish integer overflow vulnerabilities from benign overflows, but in the context of IO2BO, the involved integer overflow cannot be benign and it must be a real vulnerability.

In recent years, IO2BO is being widely exploited by attackers to cause severe damages to computer systems, such as (Chen et al. 2005; Sotirov 2007; Vreugdenhil 2020). According to statistics, from July 2019 to July 2020, the National Vulnerability Database (NVD (National Vulnerability Database 2020)) has recorded 77 IO2BO vulnerabilities, which makes up more than one third of integer overflow vulnerabilities (207 in total) recorded by NVD in the same period.

As IO2BO vulnerabilities have become a dominant kind of integer overflow vulnerabilities in practice, a variety of solutions have been proposed for IO2BO detection. The solutions can be categorized into approaches based on static analysis (e.g. Wang et al. (2012, 2009) and those depended on dynamic testing (e.g. (Zhang et al. 2010; Dietz et al. 2012; Wang et al. 2010; Chen et al.

*Correspondence: xulili@iie.ac.cn
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
Full list of author information is available at the end of the article

2012)). Dynamic testing approaches are commonly used during software deployment, but their efficiency highly relies on the completeness of the test inputs. Static analysis approaches do not require the availability of test inputs and usually take all possible paths of the programs into consideration, which makes them more populate in practice. However, the key limitation of static analysis approaches is that the reported integer overflow vulnerabilities contain too many false positives due to the lack of execution information. To solve this problem, tools such as PREfix+Z3 (Moy et al. 2009) and IntScope (Wang et al. 2009) employ symbolic execution (Brummayer 2009) to generate path constraints and prune infeasible paths in order to alleviate false positives. However, these tools may suffer from path explosion when applied to large programs. Another state-of-the-art tool, KINT (Wang et al. 2012), also collects path constraints to prune false positive but it only generates intra-procedural path constraints for each reported integer overflow program point. It also ignores implicit data dependencies imported by memory operations or complex data structures. Our experiments in "Evaluation" section shows that the false negatives reported by KINT remain high when detecting IO2BO vulnerabilities.

In the preliminary version (Mingjie X et al. 2018) of this paper, we proposed LAID, a static analysis framework utilizing an improved taint analysis (compared with KINT) and a lightweight approach for path constraint generation and solving to detect IO2BO vulnerabilities. Specifically, LAID first uses taint analysis to identify potential IO2BO vulnerabilities. It supports implicit data flow introduced by memory operations (load and store) and taint propagation on complex data structures, which improves the accuracy of detection. Then it applies an inter-procedural path-satisfiability analysis to filter out false positives. The analysis generates conditions under which an IO2BO vulnerability can be triggered in practice and then performs constraints solving by feeding the generated conditions into a solver. If the conditions of a potential IO2BO vulnerability cannot be satisfied, the potential IO2BO vulnerability will be filtered out as a false positive. Compared with symbolic execution that needs to model the runtime environment, LAID is more light-weight in the constraints generation, thus has less space and time consumption. An evaluation on 6 real-world vulnerabilities shows that LAID can effectively reduce false positives and false negatives in detecting real vulnerabilities.

However, LAID only considers the cases where the overflowed value is used in a buffer operation within the same function. LAID still misses some real vulnerabilities due to the incomplete considerations of the indirect calls in C programs and the cases where the BO sites are reachable inter-procedurally from the IO sites. In many C programs, function pointer and indirect calls are used to support a dynamic run-time feature. Existing approaches identify indirect-call targets based on type analysis, specifically, by matching the types of function pointers and the set of address-taken functions (whose addresses have been generated and stored). Such approaches, however, suffer from a high false-positive rate as many irrelevant functions may share the same types. Alternatively, one may opt for precise solutions (e.g., SVF (Sui and Xue 2016)) based on pointer analysis, which, however, would be unscalable when analyzing large-scale programs like a linux kernel.

In this article, we present ELAID (short for an Enhanced Light-weight and Accurate method of static IO2BO vulnerability Detection), an enhancement of LAID, to improve the effectiveness of LAID in three major aspects. (i) It proposes a precise and scalable indirect call analysis technique on top of the LLVM framework (Lattner 2012; Lattner and Adve 2004). It utilizes a two-stage approach that incorporates both a definition-based and the type-based ways to resolve indirect calls. (ii) Based on a more complete global call graph, it covers the cases where the tainted IO sites and the sensitive BO sites are reachable inter-procedurally. Thus two more real-world vulnerabilities are detected in the evaluation part. (iii) A large-scale program, namely, Linux kernel is added to the benchmark programs set to show the scalability of ELAID. We investigate 8 real IO2BO vulnerabilities in the history of Linux kernel, locate the precise IO and the BO sites, and revert the latest version of Linux kernel to a vulnerable version.

We implement ELAID as a prototype tool by integrating the above improvements into LAID. We evaluate ELAID's effectiveness and efficiency on 228 programs of the NIST's SAMATE Juliet test suite (`?SAMATElink`) and 7 real-world open-source applications each of which involves known IO2BO vulnerabilities. Our experimental results show that ELAID is capable of detecting IO2BO vulnerabilities in the real-world applications with low false positives and false negative rates. The results demonstrate that ELAID performs better than LAID in detecting IO sites that contaminate inter-procedural BO sites.

In summary, this paper makes the following main improvements:

- We propose a two-stage analysis to effectively refine indirect-call targets. Our indirect call analysis is elastic and does not introduce false negatives to existing type analysis-based approaches. It is also scalable for large-scale programs like linux kernel.
- We extend LAID by supporting the detection of the IO2BO vulnerabilities where the IO sites and the BO sites are reachable inter-procedurally.

- We implement a prototype tool ELAID and evaluated it on 228 programs of the NIST's SAMATE Juliet test suite and 7 real-world applications with 14 known IO2BO vulnerabilities. The experiment result shows that our framework can catch all harmful IO2BOs in the SAMATE suite with no false positive, and for real-world applications, it can significantly reduce the number of false positives and detect more known vulnerabilities than LAID and KINT.

The rest of this paper is organized as follows. Our system overview is shown in "System overview" section. In "Indirect call analysis" section we present our indirect call analysis approach. In "Identify potential iO2BO vulnerabilities" section, we describe how to use taint analysis to identify potential IO2BO vulnerabilities. In "Vulnerability filter" section, we describe how to use constraint solving to filter out potential IO2BO vulnerabilities that cannot be triggered. "Evaluation" section shows the experiment results. Related work and conclusion are discussed in "Related work" section and "Conclusions" section, respectively.

System overview

In this section, we describe the architecture of our system as illustrated in Fig. 1. It takes LLVM intermediate representation (IR) as input, which is obtained by compiling C source code using Clang, and performs a two-stage analysis to detect IO2BO vulnerabilities in these IRs. At the end of analysis, it outputs the detected IO2BOs along with their locations. The first stage constructs a whole-program call graph. An inter-procedural taint analysis is performed based on the system-wide call graph. Sensitive

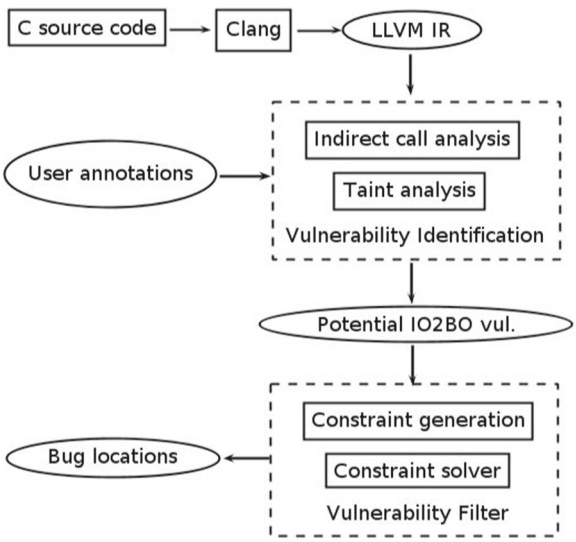

Fig. 1 The structure of our framework

taint sources and sinks are given by our built-in annotations as well as user annotations. Potentially vulnerable integer operations are identified by the comparison with the pattern of IO2BO vulnerabilities; the second stage performs constraint generation and solving for each potential IO2BO vulnerability, if the constraint of a potential IO2BO vulnerability cannot be satisfied, the vulnerability will be filtered out, which helps reduce false positives.

Next we use a real IO2BO vulnerability existed in Jbig2dec (a JBIG2 decoder library) showed in Fig. 2 as an example to explain how our system works. The IO2BO vulnerability (CVE-2016-9601) occurs in function `jbig2_image_new` at line 56 (highlighted in the red color) of jbig_image.c. The addition operation at line 56 overflows and results in a memory allocation less than expected.

Call Graph Construction. To support inter procedural analysis, we need a system-wide call graph. It is straightforward to decide the targets for direct calls as the callsite at line 129 in Fig. 2. While for C language to achieve an abstract and modular design, function pointers and indirect calls are often used. For these cases, we make use of a two-stage indirect call analysis: a definition-based way as well as a type-based way to resolve indirect calls.

Taint Analysis. Find and annotate the taint source (untrusted input) occurred in Jbig2dec, such as the parameters of main function and the pointer to file returned by `fopen` function. Perform taint tracking to determine which values can be influenced by untrusted inputs, and which values may be used in memory allocation operations, such as function `jbig2_new` in this case.

Vulnerability Identification. After the taint analysis, the parameter `width`, parameter `height` and variable `check` in function `jbig2_image_new` are found all influenced by untrusted inputs. The result of the addition operation `check+1` at line 56 is used for determining how much memory to allocate by calling `jbig2_new` function, so this line is identified as a potential IO2BO vulnerability

Vulnerability Filter. We perform constraint generation and solving for this potential IO2BO vulnerability to verify whether it can be triggered in program's execution. The constraint can be divided into overflow condition and path constraint:

Overflow Condition. If the involved integer operation overflows, the condition $(int)check + 1 > \text{INT\_MAX}$ should be satisfied.

Path constraint. We generate the path constraints from the caller of `jbig2_image_new` to the integer overflow point. Usually one function has more than one caller. As long as the path from one caller is satisfiable, we consider the potential IO2BO site as feasible. We take one of `jbig2_image_new`'s callers, namely, function `jbig2_decode_pattern_dict` as an example,

```
//jbig_image.c
34. Jbig2Image *
35. jbig2_image_new(Jbig2Ctx *ctx, int width, int height)
36. { /*width and height are from user space*/
37. Jbig2Image *image;
38. int stride;
39. int64_t check;
41. image = jbig2_new(ctx, Jbig2Image, 1);
...
47. stride = ((width - 1) >> 3) + 1;
49. check = ((int64_t) stride) * ((int64_t) height);
50. if (check != (int)check) {
...
54. }
56. image->data = jbig2_new(ctx, uint8_t, (int)check + 1);
/* integer addition overflow */
...
69. }
```

```
//jbig2_halftone.c
119. static Jbig2PatternDict *
120. jbig2_decode_pattern_dict(Jbig2Ctx *ctx, Jbig2Segment *segment,
121. const Jbig2PatternDictParams *params, const byte *data, const
     size_t size, Jbig2ArithCx *GB_stats)
122. {
123. Jbig2Image *image = NULL;
...
129. image = jbig2_image_new(ctx, params->HDPW * (params->GRAYMAX + 1),
     params->HDPH);
...
175. }
```

Fig. 2 A real-world IO2BO vulnerability in Jbig2dec

which is shown in the lower part of Fig. 2. The control flow condition from its function entry to the callsite of `jbig2_image_new` (highlighted in the blue color) is always true. Furthermore, in `jbig2_image_new`, if the program executes to the overflow point, it should satisfy the branch condition at line 50, i.e., $check == (int)check$. So the complete constraint is

$$((int)check + 1 > \text{INT\_MAX}) \wedge true \wedge (check == (int)check) \quad (1)$$

The SMT solver is invoked to decide whether or not the above constraint can be satisfied, if not, we treat the potentially vulnerable site as a false positive and hence filter it out, otherwise we report it as an IO2BO vulnerability.

Indirect call analysis

To support inter-procedural analysis, global call graph is required. To eliminate false negatives, ELAID must conservatively identify all potential targets of indirect calls. To improve the resolution rate of indirect callsites, and at the same time, to avoid the inclusion of too many false targets, we propose a two-stage indirect call analysis approach. We first use a definition-based analysis to precisely resolve indirect callisites. For the cases in which the first way fails to resolve, we turn to the type-based analysis.

For better illustrating the two-stage indirect call analysis, we take several code snippets, i.e., Figs. 3–6, from swftools-0.9.2, a tool for Adobe Flash files (SWF files). It supports SWF files creation from different kinds of content (like images, sound files, videos or sourcecode). To achieve a modular design, it relies on function pointers to define a set of abstraction layers that specify the common interfaces to different concrete implementations.

Stage-1: definition-based analysis

Function pointers used in C programs are often defined in a field of a function pointer type in a `C struct`: e.g., `func` of `struct arguments` at line 4 in Fig. 3. Many `C` interfaces (i.e. `C structs`) are *statically* allocated and initialized as the case of `func` with `c_swf`, `c_image` and `c_movie` at lines 9-11 in Fig. 3. Some interfaces may be *dynamically* allocated and initialized at run time for reconfiguration, as the case of `d->lineTo` with `polydraw_lineTo` and `linedraw_lineTo` at line 6 and 13, respectively, in Fig. 4.

For the former, ELAID scans LLVM IR linearly to find all statically allocated and initialized `struct` objects with function pointer fields. Then, for each `struct` objects, ELAID keeps tracks of which function address is assigned

```
1. typedef int command_func_t(map_t*args);
2. static struct {
3.    char*command;
4.    command_func_t* func;
5.    char*arguments;
6. } arguments[] =
7. {
8.   ...
9.   {"swf", c_swf, "name filename as3name="},
10.  {"png", c_image, "name filename"},
11.  {"movie", c_movie, "name filename"},
12.  ...
13. }
```

Fig. 3 E.g. initialization of function pointer in a struct field

```
1. void polydraw_lineTo(gfxdrawer_t *d, gfxcoord_t _x, gfxcoord_t _y);
2. void linedraw_lineTo(gfxdrawer_t *d, gfxcoord_t x, gfxcoord_t y);
3. void gfxdrawer_target_poly( gfxdrawer_t *d, double gridsize)
4. {
5.   ...
6.   d->lineTo = polydraw_lineTo;
7.   ...
8. }
9. ..
10. void gfxdrawer_target_gfxline(gfxdrawer_t*d)
11. {
12.   ...
13.   d->lineTo = linedraw_lineTo;
14.   ...
15. }
```

Fig. 4 E.g. definitions of function pointer in a struct field

to which function pointers field using an offset as a key for the field. For example, for the function pointer `func` at line 4 in Fig. 3, we use "`struct.arguments.1`" to denote that `func` is the first[1] field of `struct arguments` and its possible initialization values `c_swf`, `c_image`, `c_movie` at lines 9-11 are inserted to `func`'s possible targets set. This procedure can be intuitively expressed by the following pseudocode.

$$Callees[func] = Callees[struct.arguments.1] \\ += \{c\_swf, c\_image, c\_movie\};$$

By denoting the specific offset of the field in a struct, ELAID implements a field-sensitive analysis, it allows the collected targets to be associated with the individual field of an interface.

For the latter dynamically initialized interfaces, ELAID performs a data flow analysis to collect any assignment of a function address to the function pointer inside an interface. There are mainly three cases in which function address assignments happen:

1 *The definitions of function pointers in struct fields.* Take the function pointer `lineTo` occurring at lines 6 and 13 in Fig. 4 as an example, analogous to the case of global struct's initialization, we denote it by "`struct.gfx drawer_t.4`" where the number 4 is the index of `lineTo` field in `gfxdrawer_t`. Because of the assignments at lines 6 and 13, funtions `polydraw_lineTo` and `linedraw_lineTo` are inserted to `lineTo`'s possible targets set. We use the following pseudocode to illustrate this procedure.

$$Callees[lineTo] = Callees[struct.gfxdrawer\_t.4] \\ += \{polydraw\_lineTo, linedraw\_lineTo\};$$

2 *The definitions of global function pointer variables.* Take the function pointer `SWF_error` at line 1 in Fig. 5 as an example, because of the definition at line 6, function `print_error` is inserted to `SWF_error`'s possible targets set, which can be expressed by the following pseudocode.

$$Callees[SWF\_error] += \{print\_error\};$$

3 *Function pointers as arguments at Callsites.* Take the function pointer `callback` at line 1 in Fig. 6 as an example, we denote it by `arg.enumerateUsed IDs.2`, to represent that it is the second argument of the function call `enumerateUsedIDs`. Functions `callbackCount` and `callbackFillin` are `callback`'s possible targets, as they are passed as the actual arguments at `enumerateUsedIDs`'s callsites at lines 7 and 13, respectively. This can be represented by the following pseudocode.

$$Callees[callback] = Callees[arg.enumerateUsedIDs.2] \\ += \{callbackCount, callbackFillin\};$$

After the collection of indirect call targets at initialization sites, function pointer related definition sites and argument passing at callsites, ELAID stores the result of the above first pass in a key-value map data structure in which the key is a tuple of assignment type ("`struct`" or "`arg`"), interface type name and an offset (the index of a field in a `struct`, or the index of a function pointer argument at a callsite), and the value is a set of call targets. At each indirect callsite, ELAID retireves the type of an interface and the offset from LLVM IR, looks up the map using them as a key, and figures out the matched call targets.

Stage-2: type-based analysis

Due to the weakly-typed nature of C, e.g., prevalent uses of void pointer type and omnipotent character pointer type, when a struct-type is cast to/from or stored to general pointer types (e.g., char*) and integer types, we say that the struct-type is escaping (Lu and Hu 2019). When the previous definition-based type analysis is not available due to type escaping, ELAID can turn to a more permissive way for finding indirect call targets. Namely, we first collect the address-taken functions, and use the type-analysis-based approach (Niu and Tan 2014; Tice et al. 2014) to find the targets of indirect calls. In practice, LLVM can check which functions are-address taken, i.e.,

[1] Here the count starts at zero.

```
1. void (*SWF_error)(const char *msg, ...);
2. void print_error(const char*format,...);
3. int compileSWFActionCode(const char *script, int version, void**data
     , int*len)
4. {
5.    ...
6.    SWF_error = print_error;
7.    ...
8. }
```

Fig. 5 E.g., definition of global function pointer variable

there is a source line that generates the address of the function and stores it. Then, as long as the type of the arguments of an address-taken function matches with the callsite of the indirect call, we assume it is a valid target. When doing type matching for arguments, note that we also assume universal pointers (e.g., char *, void *) and an 8-bytes integer can match with any type.

Identify potential iO2BO vulnerabilities

The input of this module is the LLVM intermediate representation (IR) translated using clang from source code. First we annotate taint source and taint sink on LLVM IR. Second, we do taint propagation in a manner similar to classic dataflow analysis. We add the support for implicit dataflow caused by the load or store operation on the same memory address, and implement field-sensitive taint propagation for complex data structures, that is, the taint information can be propagated on specific fields of a Struct data type, which improves the accuracy of taint analysis. If any operand of an arithmetic operation is tainted (thus untrusted) and the result is used in a memory allocation function, there exists a potential IO2BO vulnerability. After the taint propagation, some candidate IO2BO vulnerabilities are generated and we collect them for further filtering.

If any operand of an arithmetic operation is tainted (thus untrusted) and the result is used in a memory allocation function, we identify it as a potential IO2BO vulnerability.

In the preliminary version of this paper, we only considered the case in which the integer overflow(IO) site and the buffer overflow(BO) site occur with in the same function. In this paper, we extend the previous work by considering also the BO sites that can be reached interprocedurally from IO sites.

Taint source initialization

Taint source represents the untrusted input of the program, which can be files, net-work data, input messages of mouse and keyboard. Generally, it is necessary to provide untrusted input source information according to the specific program under analysis. In the experiments, we annotate the parameters of the main function, the file pointer returned by the fopen function, the pointer to the buffer used in fread function, etc. as the taint source. For the case of Linux kernel, we additionally annotate "copy_from_user" and data parameters of IOCTL related functions as taint sources.

Taint propagation

Given the information of taint source, taint propagation is performed according to the algorithm shown in Algorithm 1. Since our implementation is based on LLVM IR, the algorithm mainly describes the strategies of taint propagation for several typical instructions in IR. Given the LLVM bytecode of program P, the algorithm starts with the provided taint source, propagates the tainted data and records the instructions influenced by dirty data. Finally it annotates all tainted instructions by adding metadata information on LLVM IR for corresponding instructions and outputs the modified LLVM bytecode as P'.

In Algorithm 1, we first generate a system-wide call graph for program P. To increase the precision of the analysis, we use the method introduced in

```
1. void enumerateUsedIDs(TAG * tag, int base, void (*callback)(TAG*,
     int, void*), void*callback_data);
2. void callbackCount(TAG * t,int pos, void*ptr);
3. void callbackFillin(TAG * t,int pos, void*ptr);
4. int swf_GetNumUsedIDs(TAG * t)
5. {
6.    ...
7. enumerateUsedIDs(t, 0, callbackCount, &num);
8.    ...
9. }
10. void swf_GetUsedIDs(TAG * t, int * positions)
11. {
12.    ...
13. enumerateUsedIDs(t, 0, callbackFillin, &ptr);
14. }
```

Fig. 6 E.g., function pointer as an argument at a callsite

Algorithm 1 Taint propagation algorithm

```
1: procedure TAINTPROPAGATION(Program P, Program P′)
2:     Compute a call graph CG of P, mapping a callsite to all potential callees;
3:     Let Δ be the set of tainted instructions in P;
4:     Δ = annotateTaintSource();
5:     bool changed = true;
6:     while changed do
7:         changed = false;
8:         for each instruction in each function of each module in P do
9:             if it is a memory instruction then
10:                changed |= memory_propapgation(Δ);
11:            else if it is a function call instruction then
12:                Obtain the callee according to the constructed CG;
13:                changed |= call_propapgation(Δ);
14:            else
15:                changed |= other_propapgation(Δ);
16:    P′ = P by adding to each instruction in Δ a new metadata indicating that this instruction is tainted;
```

"Indirect call analysis" section to resolve the indirectly called functions that a function pointer may point to. We use the symbol Δ to denote the set of instructions tainted by untrusty input, which is initialized to the taint source annotated by users. The main algorithm consists of a while loop, taint propagation rule is applied to each instruction in P iteratively and newly tainted instructions are added to Δ. Once there is no newly tainted instruction in an iteration, i.e., the flag variable changed is false, the process will terminate.

For different kinds of instructions, different taint propagation rules are applied. We divide the instructions into three categories: memory related instructions, function call instructions and other instructions, corresponding to the *memory_propapgation*, *call_propapgation* and *other_propapgation* sub-processes respectively in Algorithm 1.

The taint propagation strategies for these three categories of instructions are described in Table 1 below, specifying the taint status for data derived from tainted or untainted operands. Since taint can be represented with a bit, propositional logic is usually used to express the propagation policy, $T1 \vee T2$ indicates that the result is tainted if $T1$ or $T2$ is tainted.

The meaning of the taint propagation strategies in Table 1 are described in detail below:

Memory Instructions. The memory instructions include **store**, **load** instruction and the closely related **getelementptr** instruction which computes positions in a Struct data type.

Store and Load operation. For store instruction: **store** *val*, *ptr*, the *val* operand is the value to be stored and the *ptr* operand specifies the address at which to store val. The rule $T(ptr) = T(val)$ says that if the *val* operand is tainted, the *ptr* operand is set to be tainted. For load instruction: *val* = **load** *ptr*, the rule $T(val) = T(ptr)$ means that if the *ptr* operand that specifies the memory address from which to load is tainted, the *val* operand is set to be tainted.

Getelementptr operation. Getelementptr instruction: *resptr* = **getelementptr** [*struct*] .[*ptr*] .[*idx*], is used to get the address of the *idx*-th sub-element of the *struct* type pointer variable *ptr*. The first rule $T(resptr) = T(ptr) \vee T(ptr + idx)$ means that if the *ptr* operand is tainted or the address of the *idx*-th sub-element is tainted, the *resptr* is set to be tainted.

In fact, the getelementptr instruction performs address calculation only and does not access memory. Notice that the result of getelementptr instruction is usually used in load and store instructions. In order to realize a field-sensitive taint propagation for complex data structures, namely, to propagate taint data on specific fields of a Struct type, we need to record the taint status of the specific sub-element obtained through the getelementptr instruction. Essentially, the second rule $T(ptr + idx) = T(resptr)$ implies that if the address *resptr* obtained by the getelementptr instruction has been used to store dirty data, the corresponding sub-element's address $ptr + idx$ is also set to be tainted.

In Example 1 we illustrate intuitively how the taint propagation strategy for memory-related instructions works using the toy sample code in Fig. 7 below.

Example 1 *The code in Fig. 7 is a snippet of LLVM IR omitting type information for the sake of readability. It first uses getelementptr instruction to get the address of variable bar of struct TEST pointer x, namely, a*1*. Tainted data is then stored into a*1*. The address of variable bar in Struct TEST pointer x is calculated again as a*2*. The value b*1 *is read from address a*2 *and used in malloc for determining how much memory to allocate.*

The variable taint_data is initialized to be tainted, by applying the aforementioned taint propagation strategy, the taint propagation process works as shown in Table 2.

*The result of the taint propagation shows that the operand b*1 *is tainted. Since the tainted b*1 *is used in memory allocation function malloc, we conclude that there exists a security risk.*

Function Call Instructions. Call instruction represents a simple function call. We divided the function calls into programmer-defined function calls and special library function calls.

Table 1 The strategies of taint propagation for different kinds of instructions

Instruction type	Intermediate representation	Strategy T
Memory instructions	**store** *val*, *ptr*	$T(ptr) = T(val)$
	val = **load** *ptr*	$T(val) = T(ptr)$
	resptr = **getelementptr** [*struct*] .[*ptr*] .[*idx*]	(1) $T(resptr) = T(ptr) \vee T(ptr+idx)$ (2) $T(ptr+idx) = T(resptr)$
Function Call Instructions	*retval* = **call** fun(*arg*) //Definition of fun function: **define** fun(*arg_fun*) { · · · **ret** *retval_fun* }	(1) $T(arg\_fun) = T(arg)$ (2) $T(retval) = T(retval\_fun)$
	//special library function call, e.g.: *retval* = **call** fopen(*pathname*, *mode*)	$T(retval) = T(pathname)$
Other instructions	$res = \mathbf{OP}\ op_1, op_2, \ldots, op_n$	$T(res) = T(op_1) \vee T(op_2) \vee \cdots \vee T(op_n)$

Programmer-defined function. The form is *retval* = **call** fun(*arg*), where the definition of fun is: **define** fun(*arg_fun*) { · · · **ret** *retval_fun* }. The first rule $T(arg\_fun) = T(arg)$ implies that if the actual parameter of called function is tainted, the formal parameter of called funtion is set to be tainted. Similarly, the second rule $T(retval) = T(retval\_fun)$ indicates that if the return value of called funtion is tainted, the result of function call instruction is tainted.

Special library function. Thanks to the special effects of some library functions, we can directly determine the taint status of their return values or certain actual parameters. Take function fopen as an example, *retval* = **call** fopen(*pathname*, *mode*), parameter *pathname* indicates the file to be opened, parameter *mode* indicates the file access mode and *retval* is a pointer to the opened file. If *pathname* is tainted, *retval* is set to be tainted.

Other Instructions. The form is *res* = **OP** $op_1, op_2, \ldots, op_n$, such as **add**, **sub** and **mul** instructions in LLVM IR. The rule $T(res) = T(op_1) \vee T(op_2) \vee \cdots \vee T(op_n)$ says that if any operand of the instruction is tainted, the return value is set to be tainted.

```
1: a1 = getelementptr [TEST].[x].[2]
2: store taint_data, a1
3: a2 = getelementptr [TEST].[x].[2]
4: b1 = load a2
5: call = call malloc(b1)
//variable x is a pointer to Struct TEST,
//the definition of TEST is as follow.
struct TEST
{
          unsigned int foo1;
          unsigned int foo2;
          unsigned long bar;
}
```

Fig. 7 Sample code snippet for memory operation

Vulnerability identification

After annotating taint source and taint propagation, all values influenced by taint source will be marked as tainted. We identify the instructions that satisfy the following 3 conditions as potential IO2BO vulnerabilities:

1 The instruction is an integer arithmetic operation;
2 The instruction is influenced by taint source;
3 The result of the instruction is used in memory allocation function such as malloc for determining how much memory to allocate.

Example 2 *Figure 8 shows a snippet of code of LLVM IR omitting type information and type conversion for the sake of readability. It first uses alloca to allocate 16 bytes memory for variable buf. A file f*0 *is opened and the first 16 bytes of the file is stored into the memory pointed by buf. Lastly, the content of the second byte in buf is multiplied by 4 and the result is used in malloc for determining how much memory to allocate.*

*Assuming that the opened file is provided by user, the return value f*0 *is thus tainted, and the argument buf used in fread function is also set to be tainted. The variable conv obtains the contents pointed by the second byte of the buf, so conv is set to be tainted. All tainted variables are denoted in blue in Fig. 8.*

The variable mul is the result of multiplying conv by 4, and it is used in malloc, i.e., the variable mul is affected by taint source and is eventually used in memory allocation.

According to the above identification principles, the instruction mul = ***mul*** *conv*, 4 *at line 6 (highlighted in red)*

Table 2 An example of detailed taint propagation process

Step	IR Line No.	Taint propagation	Applied rule
Initialization	-	T(taint_data)=true	Initialization
The first loop	2	T(a1) = T(taint_data)	Store rule
The second loop	1	T(x->bar) = T(a1)	Getelementptr rule 2
	3	T(a2) = T(x->bar)	Getelementptr rule 1
	4	T(b1) = T(a2)	Load rule

```
1: buf = alloca [16 x i8]
2: f0 = call fopen(filename, mode)
3: call0 = call fread(buf, 1, 16, f0)
4: arrayidx = getelementptr [16 x i8].[buf].[1]
5: conv = load arrayidx
6: mul = mul conv, 4 # This line is marked as a candidate IO2BO
    vulnerability
7: call = call malloc(mul)
```

Fig. 8 Sample code for illustrating potential IO2BO vulnerabilities identification

will be marked as a potential IO2BO vulnerability and used for further filtering.

In the above example, the IO site and the BO site occur in the same function. To detect the cases where the BO sites are reached from IO sites after several function calls, we again utilize the taint analysis approach, consider the detected tainted IO sites as new tainted sources, propagate them inter-procedurally and check if they will reach any sensitive BO sites.

Vulnerability filter

After the taint analysis, the candidate IO2BO vulnerabilities are generated. However the taint analysis is path-insensitive, thus there may be many infeasible paths which bring false positives. To eliminate the false positives, we examine whether the overflow conditions under which an integer overflow may occur, and the path constraints that associated with the paths from caller functions' entry points, going through overflow points and reaching the corresponding sinks could both be satisfied. Given a candidate IO2BO vulnerability, the tactic validates if it is genuine as follows:

1. The overflow condition is calculated for the overflow point according to Table 3.
2. The path constraint encodes the conditions on both the paths from caller functions' entry points to the overflow point, and the paths from the overflow point to the corresponding sinks, also known as *forward vulnerable paths.*
3. A whole constraint formula, denoted by Π, is obtained by integrating the overflow condition and the path constraint with a logical conjunction.
4. The SMT solver is invoked to solve whether or not Π can be satisfied. If not, then the vulnerability is a false positive and hence filtered out.

Overflow condition

An N-bit signed integer is in the bounds -2^{N-1} to $2^{N-1}-1$ and an N-bit unsigned integer is in the bounds 0 to 2^{N-1}. Table 3 lists the requirements of producing an out-of-bounds result for each integer operation. The second column indicates the operands are unsigned or signed. The operands are all N-bit width integers by default. Taking division operation as an example, the divisor should be non-zero and the signed division $-2^{N-1}/-1$ is not in bounds, because the expected mathematical result 2^{N-1} is out of the bounds of N-bit signed integers.

Path constraint

We refer to the function where the integer overflow locates as the defective function and refer to the function that calls the defective function as the caller function. When generating path constraint, we collect the conditions on paths starting from the entry point of the caller function, passing through the overflow point and reaching the memory allocation operation in the defective function where the overflowed integer is used.

We do not consider the conditions on paths originating from the entry point of the whole program under test. In "Evaluation" section we will illustrate by experiments that, when considering the whole path constraints for real-world programs, the time consumption of constraints solving increases but the effectiveness of false positives filter is not improved significantly. This basically shows the fact that for heavy-weight programs, the conditions really affecting the existence of an integer overflow are usually imposed within the defective function and its caller function. We believe that for light-weight programs, considering whole path constraints can impact more on the filter. We leave it as future work to design a strategy to strike a balance between performance and low false positive rate.

The path constraint is divided into two parts for construction, intra-procedural path constraint and one-level inter-procedural path constraint.

1. First, we generate the conditions on paths within the defective function, namely, the paths starting from the defective function's entry point, passing through the risky integer operation and reaching the corresponding sinks. We denote this constraint by *IntraPC.*
2. Then we generate the conditions on paths within the caller function, namely, the paths starting from the

Table 3 Overflow condition for integer operations

Integer operation	Sign	Overflow condition
$x+y$	$<S,S>$	$x+y \notin [-2^{N-1}, -2^{N-1}]$
	$<U,U>$	$x+y \notin [0, -2^{N-1}]$
$x-y$	$<S,S>$	$x-y \notin [-2^{N-1}, -2^{N-1}]$
	$<U,U>$	$x-y \notin [0, -2^{N-1}]$
$x \times y$	$<S,S>$	$x \times y \notin [-2^{N-1}, -2^{N-1}]$
	$<U,U>$	$x \times y \notin [0, -2^{N-1}]$
x/y	$<S,S>$	$(y=0) \vee (x=-2^{N-1} \wedge y=-1)$
	$<U,U>$	$y=0$
$x << y, x >> y$	$<S,S>$	$y \notin [0, N-1]$
	$<U,U>$	

caller's entry point to the callsite of the defective function. We denote this constraint by *InterPC*.

3 We denote by *ParamPassing* the equality relations between the actual parameters in the caller function and the formal parameters in the defective function.

4 Thus, the complete path constraint is composed of:

$$IntraPC \wedge InterPC \wedge ParamPassing.$$

These constraints *IntraPC* and *InterPC* arise from two sources: assignments to variables involved in the integer operation and conditional branches along the execution path. We denote by $PC(start, end)$ the constraint on paths from basic block(BB) *start* to BB *end*, which is caculated as follows:

$$PC(start, end) = \begin{cases} True; & \text{if start = end} \\ \vee_{p \in Pred(end)} (PC(start, p) \wedge br \wedge as); & \text{otherwise} \end{cases}$$

where p is a predecessor BB of *end*, *br* is the branch condition from p to BB *end*, and *as* are the assignments to variables along the path from p to *end*. We denote by *dEntry* the entry BB of the defective function, by *io* the BB where the integer overflow point locates and by *mem* the BB where the risky memory operation locates, then $IntraPC = PC(dEntry, io) \wedge PC(io, mem)$. Analogously, we denote by *cEntry* the entry BB of the caller function and by *cs* the BB where the callsite of the defective function locates, then $InterPC = PC(cEntry, cs)$.

In the following Example 3 we use a code snippet in Fig. 9 to explain how the vulnerability filter module works.

Example 3 *According to the principles presented in "Identify potential iO2BO vulnerabilities" section, in Fig. 9 the variable n in function foo is influenced by user input, and the result of $n * 8$ is used in malloc, so the line highlighted in red is identified as a potential IO2BO vulnerability. In this example, the function foo is the defective function and the function bar is the caller function.*

```
Struct test
{
        unsigned long x;
        char ch;
};
int foo(Struct *t)
{
        unsigned long n = t->x;
        if (n > 1<<30)
                return ERROR;
        void *p = malloc(n * 8);
        /* integer multiplication overflow */
        ...
}
void bar()
{
        Struct test *s = /* from user space */;
        if (s->x >= 0 && s->x <=100)
                foo(s);
}
```

Fig. 9 Code snippet for illustrating how filter module works

The overflow condition for the argument used in malloc is $n > MAX/8$. In foo, the malloc operation will be executed under the condition (intra-procedural path constraint) that ($n <= (1 << 30) \wedge n = t->x$). In bar, the callsite to foo will be executed under the condition (the one-level inter-procedural path constraint) ($s->x >= 0 \wedge s->x <= 100$). The actual parameter s passed to foo function is equal to the formal parameter t in foo function prototype. Thus, the parameter passing condition is $s->x = t->x$. The complete path constraint is a conjunction of the previous three parts. The whole constraint Π is a conjunction of the complete path constraint and the overflow condition. Lastly, Π is fed into the solver to see if it is satisfiable. In this case, it is impossible to satisfy $n > MAX/8$ and $n <= 100$ at the same time, so the candidate vulnerability is a false positive and hence filtered out. The detailed constraints are shown in Table 4.

In (Wang et al. 2012) the algorithm that computes the path constraint for a basic block within a function was given and is shown in Algorithm 2. Here we generalize it by combining the path constraint in the caller function and the one in the defective callee function, and the resultant whole path constraint is fed to constraint solver.

Table 4 The condition of the code snippet in Fig. 9

Condition type	Content of condition
Overflow condition	$n > MAX/8$
Intra-procedural path constraint (*IntraPC*)	$n <= (1 << 30) \wedge n = t->x$
One-level inter-procedural path constraint (*InterPC*)	$s->x >= 0 \wedge s->x <= 100$
Parameter passing condition (*ParamPassing*)	$s->x = t->x$
The whole constraint Π	$(n > MAX/8) \wedge (n <= 1 << 30 \wedge n = t->x) \wedge (s->x >= 0 \wedge s->x <= 100) \wedge (s->x = t->x)$

Algorithm 2 Path constraint generation

```
1: procedure PathConstraint(blk)
2:     if blk is entry then
3:         return true
4:     g ← false
5:     for all pred ∈ blk's predecessors do
6:         e ← (pred, blk)
7:         if e is not a back edge then
8:             br ← e's branching condition
9:             as ← ∧_i(x_i = y_i) for all assignments along
   e
10:            g ← g ∨(PathConstraint(pred) ∧ br ∧
   as)
11:    return g
```

Note that for the filter process, we use a safe approximation based on the result of constraint solver. Specifically, only when the solver can determine that a possibly vulnerable IO site is not feasible, do we filter this site out, otherwise the site is kept as vulnerable. In this way, we would not miss real vulnerabilities.

Evaluation

We implement our framework as a prototype tool ELAID based on LLVM passes. The vulnerability identification module consists of four passes: annotation pass, call graph construction pass, taint pass and check pass. The annotation pass is used to recognize the taint source occurred in the program to be tested. The call graph construction pass generates a system-wide function call graph to support the following inter-procedural analysis. After the taint pass performing taint tracking to determine which values can be influenced by untrusted sources, the check pass will identify the potential vulnerabilities according to the IO2BO pattern. The vulnerability filter module consists of two passes: intrasat pass and intersat pass. The intrasat pass generates the overflow condition and the intersat pass generates the path constraint. A conjunction of the overflow condition and the path constraint is fed into the SMT solver Boolector, which provides APIs for conveniently constructing efficient overflow detection conditions.

We have tested ELAID on 228 programs in the NIST's SAMATE Juliet test suite version 1.2 and 7 real-world open-source applications, including a large scale program like linux kernel. The evaluation was performed on Ubuntu14.04 virtual machine with 32GB memory and 4 processors of an Intel Xeon host machine.

Experiments on juliet test suite

SAMATE (`?SAMATElink`) has a suite of test bench programs in C, C++ and Java to demonstrate common security problems and presents security errors in design, source code, binaries, etc. For C/C++ code, the SAMATE's Juliet test suite version 1.2 provides 61,387 test programs for 118 different CWEs. Specially, CWE 680 describes the integer overflow to buffer overflow vulnerabilities. We choose all C programs (228 programs) in CWE 680 as our experimental subjects. Each program has a good function and a bad function. The good function demonstrates normal behavior and the bad function demonstrates a vulnerability.

We applied ELAID on all these 228 programs and had detected vulnerabilities in 152 programs. For the remaining 76 programs, the values involved in the overflowed integer expressions are either constants or those generated by random functions, and therefore are beyond the scope of IO2BO conditions mentioned in "Vulnerability identification" subsection. In other words, ELAID had successfully reported all harmful IO2BOs in the 228 programs with no false positive.

Experiments on real iO2BOs

In order to evaluate ELAID in real IO2BOs, 7 real-world open-source applications are chosen, containing 14 known IO2BO CVEs (Common Vulnerabilities and Exposures (CVE) 2020). Their information is listed in Table 5. Columns 1-4 describe the vulnerable software, version, LOC(lines of code) and the CVE number, respectively. For the case of linux kernel, we build it under the default configuration and its LOC is obtained by counting just the built source files. Columns **IO_op** and **MEM** show the type of the overflowed integer operation and the name of the risky sink, respectively.

Table 5 Information of applications used in evaluation

Programs	Version	LOC	CVE Number	IO_op	MEM
gocr	0.40	21608	CVE-2005-1141	$*_s$	malloc
jasper	1.900.1	28279	CVE-2011-4517	$*_u$	jas_malloc
cpio	2.9	30309	CVE-2014-9112	$+_u$	xmalloc
libexif	0.6.21	10828	CVE-2016-6328	$*_u$	exif_mem_alloc
jbig2dec	0.13	10750	CVE-2016-9601	$+_s$	jbig2_new
swftools	0.9.2	211618	CVE-2017-16868	$*_s$	malloc
linux kernel	5.8-rc1	1473247	CVE-2019-14283	$*_u$	memcpy
			CVE-2018-13406	$*_u$	kmalloc
			CVE-2017-8924	$-_u$	memcpy
			CVE-2016-9084	$*_s$	kzalloc
			CVE-2016-3135	$+_u$	kvmalloc
			CVE-2014-9904	$*_u$	kmalloc
			CVE-2012-6703	$*_u$	kmalloc
			CVE-2012-0044	$*_s$	kzalloc

Evaluation of indirect call analysis We compared the effectiveness and efficiency of ELAID's indirect call analysis with the definition-based approach and with the type-based approach. LAID from the preliminary version of this paper resolves indirect calls based on merely the definition-based approach. Table 6 summaries evaluation results of ELAID, comparing it to the definition-based approach and the type-based approach in terms of the percentage of indirect callsite(ICS for short) resolved and the average number of targets per ICS.

For these tested programs, they contain 1.08% percentage of ICS in average. LAID from the preliminary version of this paper resolves in average 78% of all indirect callsites, which is lower than that of ELAID, i.e., 88.7%. For ELAID, the number of average indirect call targets per resolved indirect callsite is 16.3. It is smaller than that of the type-based approach, i.e., 29, by 43.8% reduction, as shown in the last row of Table 6. The reason is that the type-based approach classifies all functions into different sets based on the function type. This implies that all functions in the set are regarded as possible call targets of that function pointer. Table 6 shows that ELAID can achieve a higher resolution rate than LAID(merely definition-based approach), and at the same time have a lower average number of targets per ICS than the type-based approach.

Effectiveness of Vulnerability Identification. In Table 7, we evaluate the effects of vulnerability identification with the underlying call graph generated by the definition-based, the type-based and ELAID's approaches, respectively. In Table 8, we evaluate the effects of vulnerability identification of IO2BO bugs in which the IO and BO sites are reachable intra-procedurally, and the ones in which the IO and BO sites are reachable inter-procedurally(IP), we call them IO2IPBO for short in the following.

Usually in large software, developers will encapsulate memory allocation operations. Therefore, we do not only annotate malloc, calloc, realloc, etc. as the sinks of taint analysis, but also annotate corresponding wrapped memory allocation functions in the testcases as sinks, such as "jas_malloc", "exif_mem_alloc" and "jbig2_new".

Table 7 shows, for each benchmark program, the number of total integer arithmetic operations in the program (column 2) and the number of integer arithmetic operations identified as potential IO2BO sites (column 3, 6 and 9), with the underlying call graph generated by the definition-based, the type-based and ELAID's approaches, respectively. Column 4, 7 and 8 shows the checking ratio, i.e., (the number of integer arithmetic operations identified as potential IO2BO sites) / (the total number of integer arithmetic operations). Results show that, the percentage of integer operations that need to be checked is reduced to very small, on average around 2%∼2.2%, by vulnerability identification module.

Table 8 displays, for each benchmark program, the number of the identified IO2IPBO sites and the corresponding analysis time. To tradeoff between the vulnerability detection rate and a conservative number of reports, after investigating two real IO2IPBO vulnerabilities in linux kernel (i.e., CVE-2019-14283 and CVE-2017-8924), we set in the experiment the length of call chains from IO sites to BO sites to be 3. The last two columns show the performance comparison in terms of the ratios of the increased vulnerable sites and of the increased time overhead. As shown in Table 8, by considering the BO points that can be reached inter-procedurally from IO points, i.e., IO2IPBOs, the number of identified vulnerable sites are increased by 73.2% on average. The time overhead is on average doubled as IO2IPBOs are identified by reusing the taint analysis framework.

Effectiveness of Vulnerability Filter. In this section, we evaluate the effect of filter module. The experimental results are given in Table 9. Column 1 lists the names of the benchmarks. Column 2 shows the number of potential IO2BO vulnerabilities obtained after performing LAID's vulnerability identification. Columns 3 to 5 show the number of remaining potential IO2BO sites after doing vulnerability filter of one-level inter-procedural path constraints, the filter ratio and its time usage in seconds. The filter module successfully filters out a significant portion of potential IO2BO vulnerabilities that cannot be triggered, on average, 49.2% of integer operations are filtered out. Particularly, for swftools, more than two-thirds of the suspicious points are filtered out.

As mentioned in "Path constraint" subsection, we compare the performance of filter strategies between one-level inter-procedural path constraints and whole program path constraints. Columns 6 to 8 in Table 9 show the number of remaining potential IO2BO sites after doing vulnerability filter of whole program path constraints, the filter ratio and its time usage in seconds. The last two columns of Table 9 show the performance comparison in terms of the reduction ratio in the number of filtered sites, as well as the ratio of the increased time overhead. As shown in the table, the filter module considering whole program path constraints improves the filter effect by 8.8% on average, however, it costs on average 4.1X more time on path constraints generation and solving than the filter module considering one-level inter-procedural path constraints. Particularly, in the cases of libexif, jbig2dec and swftools, no suspicious point is further filtered out. Thus, ELAID decides to generate one-level inter-procedural path constraints only.

Comparison of ELAID with LAID and KINT. LAID is a tool proposed by the preliminary version of this paper. ELAID is an enhancement of LAID by using a two-stage indirect call analysis and considering BO sites that are reachable from IO sites inter-procedurally. KINT is a well-

Table 6 Indirect Call Analysis

Programs	Total CS	Total ICS	% of ICS	LAID(Definition-based)		Type-based		ELAID(LAID+Type)	
				% of ICS resolved	# of avg target	% of ICS resolved	# of avg target	% of ICS resolved	# of avg target
gocr	4207	2	0.05%	100%	1	100%	2	100%	1
jasper	3722	56	1.5%	98%	3.6	100%	11.3	100%	3.7
cpio	2747	24	0.87%	63%	2.2	83%	2.6	83%	2.2
libexif	2450	22	0.9%	95%	2.9	95%	9.1	95%	2.1
jbig2dec	2668	30	1.12%	100%	2.9	100%	3.3	100%	2.9
swftools	62178	723	1.16%	66%	7	100%	75.5	100%	63.9
linux kernel	1160541	22632	1.95%	25%	5	43%	98.2	43%	38.4
Average	-	-	1.08%	78%	3.5	88.7%	29	88.7%	16.3

Table 7 Performance of Vulnerability Identification under Different Indirect Call Analysis

Programs	#Total-int-ops	LAID(Definition-based)			Type-based			ELAID(LAID+Type)		
		#IO2BO-sites	Ratio	Analysis time(s)	#IO2BO-sites	Ratio	Analysis time(s)	#IO2BO-sites	Ratio	Analysis time(s)
gocr	4583	23	0.5%	2.1	24	0.5%	4	24	0.5%	3
jasper	2482	84	3.4%	2.9	84	3.4%	5	84	3.4%	5
cpio	655	17	2.6%	1.1	19	2.9%	< 1	19	2.9%	< 1
libexif	597	19	3.18%	< 1	19	3.18%	< 1	19	3.18%	< 1
jbig2dec	778	10	1.29%	< 1	14	1.8%	< 1	14	1.8%	< 1
swftools	8253	233	2.7%	11.4	242	2.9%	13	242	2.9%	15
linux kernel	63739	313	0.5%	7202	325	0.5%	6682	325	0.5%	6567
Average	-	-	2%	-	-	2.2%	-	-	2.2%	-

Table 8 Comparison between IO2BO and IO2IPBO sites

Programs	IO2BO		IO2IPBO		Performance Comparison	
	#sites	Analysis time(s)	#sites	Analysis time(s)	Detection improvement	Overhead in time(s)
gocr	24	3	30	3	25%	0
jasper	84	5	157	5	87%	0
cpio	19	< 1	41	2	116%	1X
libexif	19	< 1	40	2	110%	1X
jbig2dec	14	< 1	27	5	92.8%	4X
swftools	242	15	422	19	74.4%	0.27X
linux kernel	325	6567	348	7766	7%	0.18X
Average	-	-	-	-	73.2%	0.92X

known static tool that utilizes taint analysis and constraint solving to locate and filter integer overflow. Contrary to our work, KINT attempts to denote all integer errors in a program and does not make a clear distinction between classic IO errors and IO2BO errors that constitute vulnerabilities.

To compare with KINT fairly, we annotate the same taint source and taint sink for KINT and set the same time threshold for SMT solver. Among the integer overflows reported by KINT, only those involving data from an untrusted input (source) and being used in a sensitive context (sink) are counted.

Table 10 summarizes the results of comparison. Column "Result" shows whether a tool reports the IO2BO vulnerabilities, notation "✓" means the corresponding vulnerability is detected while notation "x" means not. Column "Time1" and "Time2" in ELAID and LAID are the time usages in seconds for taint analysis and for constraint generation and solving, respectively. Column "Analysis time" in KINT is the time usage of the whole process.

The comparison experiment shows that ELAID successfully detected all the 14 IO2BO vulnerabilities under examination. LAID fails to detect CVE-2019-14283 and CVE-2017-8924, because the BO sites in these two vulnerabilities are reachable from IO sites inter-procedurally, and LAID does not consider this situation. In comparison, KINT detected just 4 vulnerabilities. The reason lies in that KINT's support for implicit data streams (such as memory-related operations load/store) and complex data structures during the process of taint propagation is not accurate enough. In this way, the vulnerable paths are missed when the taint propagation is inaccurate.

Related work

Source Code Analysis. There has been a number of tools proposed to detect integer overflow at the source code level. These approaches can be classified into two broad groups: instrumenting the source code with runtime integer overflow check (e.g. (Zhang et al. 2010; Dietz et al. 2012; Brumley et al. 2007)) and using static analysis to detect integer overflow (e.g. (Wang et al. 2012)).

RICH (Brumley et al. 2007) is a compiler-based tool that instruments programs to capture runtime overflows. It protects against many kinds of integer errors, including signedness error, integer overflow/underflow or truncation error. However, benign and unexpected overflows are not distinguished.

IOC (Dietz et al. 2012) performs a compiler-time transformation operating on the Abstract Syntax Tree (AST) to add integer overflow check. Then a runtime library is linked into the compiler's output and handles integer overflows as they occur.

IntPatch (Zhang et al. 2010) is built on top of LLVM and detects vulnerabilities utilizing the type inference on LLVM IR. If a variable involved in an arithmetic operation has an untrusted source and the respective sink may overflow, IntPatch will insert a check statement after that vulnerable arithmetic operation to catch vulnerability at runtime. However, IntPatch would produce false positives if sanitization routines are added by developers. Similar to IntPatch, IntTracker (Sun et al. 2015) instruments integer arithmetic operations to monitor overflows at runtime while integrates an efficient overflow tracking technique to bypass the false positives caused by sanitization routines troubling IntPatch.

Using tools that instrument the source code with runtime overflow check to find integer overflows requires carefully chosen inputs to trigger them. Because integer errors typically involve corner cases, these tools tend to have low coverage.

KINT (Wang et al. 2012) performs three different analyses including function-level analysis, range analysis and taint analysis on LLVM IR to detect all integer errors in a program. To avoid path explosion, KINT performs

Table 9 Performance of Vulnerability Filter

Programs	Vuln. Identif. by LAID	Vuln. Filter with one-level inter-procedural path constraint			Vuln. Filter with whole program path constraint			Performance Comparison	
		Number of remaining IO2BO sites	Filter ratio	Time (s)	Number of remaining IO2BO sites	Filter ratio	Time (s)	Filter ratio	Time (s)
gocr	23	10	56.5%	11.8	7	69.6%	23.3	13.1%	2X
jasper	84	51	39.3%	8.4	37	56%	25.3	16.7%	3X
cpio	17	13	23.5%	1.1	9	47.1%	1.7	23.6%	1.5X
libexif	19	8	57.9%	2.2	8	57.9%	9.8	0%	4.5X
jbig2dec	10	5	50.0%	1428	5	50%	1725.2	10%	1.2X
swftools	233	75	67.8%	1466	75	67.8%	18369.2	0%	12.5X
Average	-	-	**49.2%**	-	-	**58%**	-	**8.8%**	**4.1X**

Table 10 Statistics of comparison experiment

Programs	CVE Numbers	ELAID			LAID			KINT	
		Result	Time1(sec) (Vuln.Identif.)	Time2(sec) (Vuln.Filter)	Result	Time1(sec) (Vuln.Identif.)	Time2(sec) (Vuln.Filter)	Result	Analysis time(s)
gocr	2005-1141	✓	3	< 1	✓	2.1	11.8	x	1082
jasper	2011-4517	✓	5	33	✓	2.9	8.4	x	768.8
cpio	2014-9112	✓	2	1	✓	1.1	1.1	x	18.5
libexif	2016-6328	✓	2	1	✓	< 1	2.2	x	6.3
jbig2dec	2016-9601	✓	5	108	✓	< 1	1428	x	190.3
swftools	2017-16868	✓	19	363	✓	11.4	1466	x	1752
linux kernel	2019-14283	✓	7766	3210	x	6567	3353	x	5563
	2018-13406	✓			✓			x	
	2017-8924	✓			x			x	
	2016-9084	✓			✓			✓	
	2016-3135	✓			✓			x	
	2014-9904	✓			✓			✓	
	2012-6703	✓			✓			✓	
	2012-0044	✓			✓			✓	

constraint solving at the level of individual functions and statically generates a single path constraint for each integer operation. Despite substantial effort, KINT reports a large number of false positives. Compared with our system, KINT attempts to find all integer errors in a program not just IO2BO vulnerabilities. KINT only considers if the overflow point can be triggered within a function and its analysis for implicit data flow and complex data structures is not accurate enough.

Binary Analysis. Many tools have been proposed to detect overflow in binaries. Followings are some representative works.

IntFinder (Chen et al. 2009) recovers type information from binaries and creates the suspect integer bug set, then uses its implemented dynamic detection tool that combined with taint analysis to rule out false positives. IntScope (Wang et al. 2009) performs a path sensitive data flow analysis on its own IR by leveraging symbolic execution and taint analysis to identify the vulnerable point of integer overflow. To deal with false positives, it relies on a dynamic vulnerability test case generation tool to generate test cases which are likely to cause integer overflows. Both IntFinder and IntScope use static analysis to find suspicious integer overflow vulnerabilities, then dynamically check each suspicious vulnerability. However, this mechanism suffer from low efficiency because of high positives of static analysis and large time consumption of dynamically checking.

INDIO (Zhang et al. 2015) is a static analysis based framework to detect and validate integer overflow vulnerabilities in Windows binaries. INDIO integrates the techniques of pattern-matching, vulnerability ranking, and selective symbolic execution to detect integer overflow in x86 binaries. At the end of its analysis, INDIO outputs the detected integer overflow vulnerabilities, as well as example inputs to the binaries that expose these vulnerabilities.

(Chen et al. 2012) chooses suitable instructions to generate constraint dynamically with taint analysis and loop analysis by launching target program. If the constraint expression is satisfiable, an overflow vulnerability is reported. RICB (Wang et al. 2010) decompiles binaries to assembly language, locates the overflow points and checks run-time integer overflow via buffer overflow. Since RICB and (Chen et al. 2012) are dynamic analysis tools with runtime check, the effectiveness depends on the set of inputs used to execute the program.

IO2BO can be considered as a special kind of heap overflow, HOTracer (Jia et al. 2017) proposes a new offline dynamic analysis solution to discover heap vulnerabilities. It selects useful testcases for programs to generate execution traces. Then it reasons about the path conditions and vulnerability conditions built by tracking heap objects' spatial and taint attributes during execution traces to generate a PoC to find heap vulnerabilities.

Conclusions

In this paper, we present a framework that utilizes static analysis techniques to detect IO2BO vulnerabilities in source code, while significantly reduces the number of false positives being reported. It constructs a complete and global call graph using a two-stage indirect call analysis approach, applies inter-procedural taint analysis to accurately and quickly identify potential IO2BO vulnerabilities and uses light-weight constraint generation and solving to verify if an IO2BO vulnerability can be triggered in the program's execution.

A prototype tool named ELAID is implemented based on LLVM. The results of our evaluation demonstrate that our tool can work on real-world IO2BO vulnerabilities and achieve a better performance compared with the preliminary version of the tool LAID and the state-of-the-art tool KINT.

In this paper we focus on IO2BO vulnerabilities, as integer overflows in the context of IO2BO can not be benign (Zhang et al. 2010) and tend to be more exploitable. Note that our framework can be generalized to detect other types of vulnerabilities, by accordingly modifying the vulnerability condition. For example, for buffer overflow vulnerabilities, the original data's possible length must be bigger than the targeted buffer's real capacity, which constructs the buffer overflow condition.

As of future work, with the analysis results of ELAID, we plan to combine symbolic execution and fuzzing to verify the authenticity of the suspicious IO2BO vulnerabilities and construct a PoC (Proof of Concept) that can trigger the corresponding IO2BO vulnerability. This would improve the practicality of our tool and form a complete tool chain that integrates identification, filtering and verification for finding vulnerabilities.

Acknowledgments

The authors would like to thank Defang Bo for preparing partial experiments' environment and conditions.

Authors' contributions

LX proposed the technical route. LX and MX performed the experiments and drafted the paper. FL and WH made crucial contributions on the technical route and revised the article. The author(s) read and approved the final manuscript.

Funding

This research was supported in part by the National Natural Science Foundation of China (Grant No. 61802394, U1836209), Foundation of Science and Technology on Information Assurance Laboratory (No. KJ-17-110), National Key Research and Development Program of China (2016QY071405), Strategic Priority Research Program of the CAS (XDC02040100, XDC02030200, XDC02020200).

Availability of data and materials

All public dataset sources are as described in the paper.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China.

Received: 10 July 2020 Accepted: 17 August 2020
Published online: 08 September 2020

References

Brumley D, Song DX, Chiueh T, Johnson R, Lin H (2007) RICH: automatically protecting against integer-based vulnerabilities. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, 28th February - 2nd March 2007. The Internet Society, San Diego
Brummayer R (2009) Efficient smt solving for bit-vectors and the extensional theory of arrays. PhD thesis. Johannes Kepler University, Linz
Chen K, Feng D, Su P (2012) Dynamic overflow vulnerability detection method based on finite csp(in chinese). In: Chinese Journal of Computers, vol 35. Science Press, Beijing. pp 898–909
Chen P, Han H, Wang Y, Shen X, Yin X, Mao B, Xie L (2009) Intfinder: Automatically detecting integer bugs in x86 binary program. In: Information and Communications Security, 11th International Conference, ICICS 2009, December 14-17, 2009. Proceedings. LNCS, vol 5927. Springer, Beijing. pp 336–345
Chen S, Xu J, Sezer EC (2005) Non-control-data attacks are realistic threats. In: McDaniel PD (ed). Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005. USENIX Association, Baltimore
Christey S, Martin RA (2007) Vulnerability Type Distributions in CVE. http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf
Common Vulnerabilities and Exposures (CVE) (2020). http://cve.mitre.org/
CWE-680: IO2BO Vulnerabilities (2020). http://cwe.mitre.org/data/definitions/680.html
Dietz W, Li P, Regehr J, Adve VS (2012) Understanding integer overflow in C/C++. In: Glinz M, Murphy GC, Pezzè M (eds). 34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012. IEEE Computer Society, Zurich. pp 760–770
Jia X, Zhang C, Su P, Yang Y, Huang H, Feng D (2017) Towards efficient heap overflow discovery. In: Kirda E, Ristenpart T (eds). 26th USENIX Security Symposium, USENIX Security 2017, August 16-18, 2017. USENIX Association, Vancouver. pp 989–1006
Lattner C (2012) LLVM: An Infrastructure for Multi-Stage Optimization. http://llvm.cs.uiuc.edu
Lattner C, Adve VS (2004) LLVM: A compilation framework for lifelong program analysis & transformation. In: 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004. IEEE Computer Society, San Jose. pp 75–88
Lu K, Hu H (2019) Where does it go?: Refining indirect-call targets with multi-layer type analysis. In: Cavallaro L, Kinder J, Wang X, Katz J (eds). Proceedings of ACM SIGSAC Conference on Computer and Communications Security (CCS). ACM, London. pp 1867–1881
Mingjie X, Shengnan L, Lili X, Feng L, Wei H, Jing M, Xinhua L, Qingjia H (2018) A Light-Weight and Accurate Method of Static Integer-Overflow-to-Buffer-Overflow Vulnerability Detection. In: Fuchun Guo, Xinyi Huang, Moti Yung (eds). Information Security and Cryptology - 14th International Conference, Inscrypt 2018, December 14-17, 2018, Revised Selected Papers. Springer, Fuzhou Vol. 11449. pp 404–423
Moy Y, Bjørner N, Sielaff D (2009) Modular bug-finding for integer overflows in the large: Sound, efficient, bit-precise static analysis. Technical report. Technical Report MSR-TR-2009-57, Microsoft Research
(2017) National Institute of Standard and Technology (NIST). SAMATE-software assurance metrics and tool evaluation. http://samate.nist.gov/SARD/testsuite.php
National Vulnerability Database (2020). http://nvd.nist.gov/
Niu B, Tan G (2014) Modular control-flow integrity. In: O'Boyle MFP, Pingali K (eds). ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14 - June 09 - 11, 2014. ACM, Edinburgh. pp 577–587
Sotirov A (2007) Heap feng shui in javascript. https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf
Sun H, Zhang X, Su C, Zeng Q (2015) Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability. In: Bao F, Miller S, Zhou J, Ahn G (eds). Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, April 14-17, 2015. ACM, Singapore. pp 483–494
Sui Y, Xue J (2016) Svf: interprocedural static value-flow analysis in llvm. In: Proceedings of the 25th International Conference on Compiler Construction. Association for Computing Machinery, New York. pp 265–266
Sun H, Zhang X, Su C, Zeng Q (2015) Efficient dynamic tracking technique for detecting integer-overflow-to-buffer-overflow vulnerability. In: Bao F, Miller S, Zhou J, Ahn G (eds). Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, April 14-17, 2015. ACM, Singapore. pp 483–494
Tice C, Roeder T, Collingbourne P, Checkoway S, Erlingsson Ú, Lozano L, Pike G (2014) Enforcing forward-edge control-flow integrity in GCC & LLVM. In: Fu K, Jung J (eds). Proceedings of the 23rd USENIX Security Symposium, August 20-22, 2014. USENIX Association, San Diego. pp 941–955
Vreugdenhil P (2020) Pwn2Own 2010 Windows 7 Internet Explorer 8 Exploit. http://vreugdenhilresearch.nl/Pwn2Own-2010-Windows7-InternetExplorer8.pdf
Wang X, Chen H, Jia Z, Zeldovich N, Kaashoek MF (2012) Improving integer security for systems with KINT. In: Thekkath C, Vahdat A (eds). 10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, October 8-10, 2012. USENIX Association, Hollywood. pp 163–177
Wang Y, Gu D, Xu J, Wen M, Deng L (2010) RICB: integer overflow vulnerability dynamic analysis via buffer overflow. In: Lai X, Gu D, Jin B, Wang Y, Li H (eds). Forensics in Telecommunications, Information, and Multimedia - Third International ICST Conference, e-Forensics 2010, November 11-12, 2010, Revised Selected Papers. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol 56. Springer, Shanghai. pp 99–109
Wang T, Wei T, Lin Z, Zou W (2009) Intscope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution. In: Proceedings of the Network and Distributed System Security Symposium, NDSS 2009, 8th February - 11th February 2009. The Internet Society, San Diego
Zhang Y, Sun X, Deng Y, Cheng L, Zeng S, Fu Y, Feng D (2015) Improving accuracy of static integer overflow detection in binary. In: Bos H, Monrose F, Blanc G (eds). Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, November 2-4, 2015, Proceedings. LNCS, vol 9404. Springer, Kyoto. pp 247–269
Zhang C, Wang T, Wei T, Chen Y, Zou W (2010) Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time. In: Gritzalis D, Preneel B, Theoharidou M (eds). Computer Security – ESORICS 2010. Springer, Berlin, Heidelberg. pp 71–86

Publisher's Note