"Aikido: Tunable Cyber Defensive Security Mechanisms"

Aarno Labs LLC

Final Technical Report

Dec 31 2022
Sponsored By
Defense Advanced Research Projects Agency (DOD)
(Information Innovation Office (I20))

Issued by DOI, Interior Business Center
Under
Contract No. D17PC00117

Name of Contractor:	Aarno Labs LLC
Principal Investigator:	Ricardo Baratto
Business Address:	245 Main St, 2nd floor, Cambridge MA 02142
Phone Number:	681-222-7664
Contract Effective Date:	March 16 2017
Short Title:	AIKIDO
Contract Expiration Date:	December 31 2022
Reporting Period:	March 16 2017 to December 31 2022

TABLE OF CONTENTS

Section **Page**

List of Figures **iii**

1 INTRODUCTION **1**

2 METHODS, ASSUMPTIONS, AND PROCEDURES **3**
2.1 System Overview 4
2.2 Trace Instrumentation 6
2.3 Targeted Input Synthesis 13
2.4 Comfortfuzz: Orchestration and Automation 20
2.4.1 OSS-Fuzz Applications 24
2.5 Vulnerability Injection Process 29
2.5.1 Conditional Branch Modification 31

3 RESULTS AND DISCUSSION **37**
3.1 OSS-Fuzz Results 38

4 CONCLUSION **42**

List of Figures

Figure **Page**

2.1 Aikido Architecture 4
2.2 TIS Architecture 20

Chapter 1

INTRODUCTION

Evaluating cyber defense security mechanisms and techniques is a difficult and manual process. To effectively compare these, security analysts use synthetic benchmarks comprised of simple [9] or artificial test cases [4, 20], a combination of known vulnerabilities which attempt to provide coverage for different classes of attacks [7], or manually injected bugs in programs of interest [21]. As such, security mechanisms adapt their techniques to guarantee good coverage for the synthetic benchmarks but provide little guarantees of their efficacy beyond that. Furthermore, these generic test cases may fail to represent the security mechanisms' users environment and expectations. To ensure accurate evaluation and comparison of competing security techniques on real-world programs, we need automated techniques for injecting realistic and verifiable vulnerabilities.

To this end, we have developed Aikido, a new technique and system for automatically generating and injecting realistic vulnerabilities to real-world applications. Aikido is built to operate on any existing C program, allowing users to create vulnerabilities and evaluate security mechanisms on those applications that they are most interested in.

Aikido uses targeted symbolic execution to discover program paths that could be used to generate vulnerabilities. The programs paths (i.e, symbolic constraints) are then modified using information from formal methods (e.g. using SMT solvers) to generate and inject new code, at the source-level, that is provably vulnerable (e.g., the system can prove that the generated conditions along a specific program path can lead to a vulnerability). The code is then obfuscated, using previously learned bug patterns, to look like native vulnerable code. To enable Aikido to generate vulnerabilities deep inside

complex applications, a scenario that is difficult for existing symbolic execution engines to solve due to path explosion and over-constraining paths, it uses goal-directed branch enforcement [24] to select only the relevant conditions required to reach a specific program path.

Aikido is implemented as a compiler pass and runtime component in the LLVM Compiler Infrastructure [12], a set of Clang-based tools, and an orchestration and automation system written in Python. To ease integration and wide use, Aikido has built-in support for Google's OSS-Fuzz [15] system, thus allowing any project that has been set up to run there to be used as a target.

We have evaluated Aikido on both standalone, manually onboarded applications and more than 200 OSS-Fuzz applications. Aikido was able to automatically generate vulnerabilities for a number of applications. We also identified a number of important avenues for further development.

Chapter 2

METHODS, ASSUMPTIONS, AND PROCEDURES

For all the research we performed in this program, we adopted an experimental approach driven by test cases. Our overarching goal was to produce systems that could be successfully applied to real-world applications.

We initially focused on applications for which our previous research [24, 22, 23] had found vulnerabilities, and which had been patched since. Our focus was to develop a system that could revert these known patches. This ground truth enabled the rapid prototyping of the system.

During the next development stage, we manually kept track of open-source applications with reported vulnerabilities in classes supported by our techniques and which had self-contained fixes. In this manner, we broadened our ground-truth base and the support of the system to a larger set of realistic applications. During this stage, we realized that one of the major roadblocks was adapting to the idiosyncrasies of the build systems used by each different application. It is a well known characteristic of software engineering that a major part of application development is coming up with new and exciting ways in which to collect dependencies, configure the compiler, and build an application and its supporting libraries.

With this newfound knowledge in hand, we pivoted towards supporting Google's OSS-Fuzz [15] system. OSS-Fuzz created a standard interface for building open source applications which were interested in being fuzzed. By piggybacking on this interface, we were able to quickly grow the number of supported applications, allowing us to focus on the hard problems that lay beyond figuring out how to build each and every application under the sun.

During the course of the research, we devoted a major effort to the packaging, testing, and usability of the system. Our releases include both unit and system tests (comprising our regression test). Regression tests were a key component to our development approach, allowing us to easily experiment with different approaches and confirm that the system was still meeting its goals.

All of the systems that we developed run on open source infrastructure (e.g., LLVM, OSS-Fuzz) and do not require proprietary software to build and run.

2.1 System Overview

To evaluate our approach, we built Aikido, a system to automatically generate and inject realistic, source-level vulnerabilities to real-world applications.

Figure 2.1: Aikido Architecture

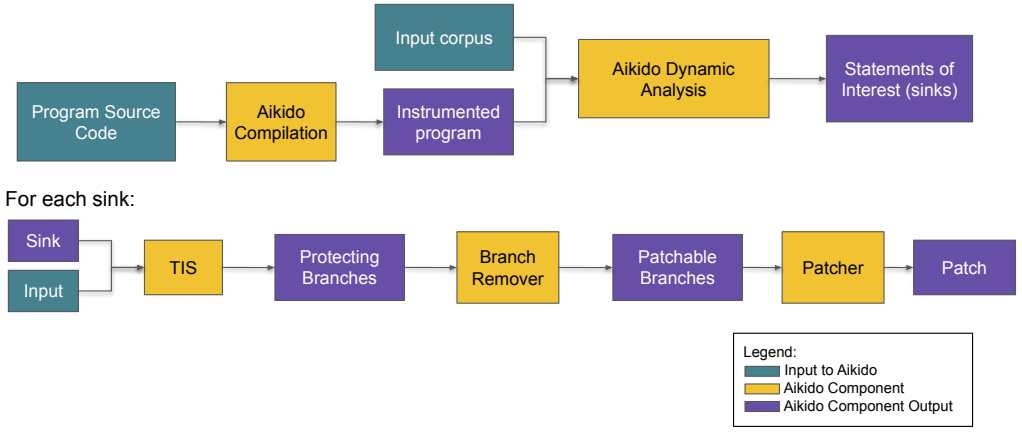


Figure 2.1 shows the mode of operation. Each of these steps will be discussed in greater detail in later sections:

- The source code for a target application is compiled to insert instrumentation which will log, at runtime, information about all operations performed by the program as it processes its input.
- The instrumented program is run over test or other inputs. For each input, an operational trace is generated. The trace is analyzed offline

and a list of target statements of interest is generated, called *sinks*. These include memory copies, array accesses, string operations, etc.

- Symbolic expressions are generated for each of the sinks in terms of the bytes in the input. The expression for each sink is passed to TIS (Section 2.3) to verify that the sink is an adequate target for the system, ie. it can be made vulnerable, but also that existing checks in the application are adequate, ie. a vulnerability does not exist already.

- In addition to verifying that a sink is an adequate target, TIS generates a list of constraints that prevent a vulnerability from existing at the sink. These constraints represent conditional branches in the source code, and they are removed one by one until a combination is found that makes the sink vulnerable.

- The branches are patched using pre-existing templates to make them vulnerable without removing them from the source code. Part of the patching process uses an SMT solver to verify the vulnerability, and TIS is used to generate inputs that trigger the vulnerability.

- The patch and input as proof of vulnerability are the final output of the system.

This approach has a number of attributes that make it superior to existing tools. First, vulnerabilities are added to existing programs. This enables testing security tools and mechanisms on real-world complexity using the natural flow of the program. In addition, new tests can easily be created, thus avoiding the pitfalls of over fitting that static benchmarks suffer from. And tests can be created over a specific corpus, such as those used, developed, or of interest to a business or government entity.

Second, the template-based system leads to vulnerabilities inserted by the system being similar to those found in real-world vulnerabilities. Third, test inputs are created as part of the vulnerability insertion process, enabling verification of the existence of the vulnerability. And finally, possible false positives can be created as well, providing a different axis of evaluation for security tools and mechanisms.

This is an example of the vulnerabilities that the system can inject. This one was generated against `jasper 1.900.1`. The original code is as follows:

```
if (JAS_CAST(int, sot->tileno) >= dec->numtiles) {
  jas_eprintf("invalid tile number in SOT marker segment\n");
  return -1;
}
```

And after the vulnerability patch is applied:

```
if (JAS_CAST(int, sot->tileno) > dec->numtiles) {
  jas_eprintf("invalid tile number in SOT marker segment\n");
  return -1;
}
```

By changing `>=` to `>` the range of inputs accepted by the program is increased, which in turn leads to a heap buffer overflow.

The following sections describe the components of the system in greater detail. Section 2.2 describes the system instrumentation. Section 2.3 describes Target Input Synthesis, the main technique used by Aikido to inject vulnerabilities. Section 2.4 describes Comfortfuzz, Aikido's orchestration and automation system. And finally, Section 2.5 describes how vulnerabilities are injected into applications.

2.2 Trace Instrumentation

Aikido's trace instrumentation is a fine-grained dynamic taint analysis built on top of LLVM's [12] DataFlow Sanitizer [5]. The instrumentation is designed to be low-overhead by combining a compilation-based approach for instrumentation and a two-step, offline analysis process that minimizes the amount of work that needs to be done as the application is running.

The purpose of the instrumentation is to generate symbolic expressions for statements of interest, sinks and branches, for a given program and input. Sinks consist of operations such as memory allocation sites (`malloc, calloc`), array index accesses, memory transfer operations (`memcpy`), string copies (`strcpy`), index operations (`strstr`) and format expansions (`sprintf`), and command execution operations (`execve, system, popen`).

The instrumentation uses taint tracing to follow input as it's processed by the application. In addition, as operations of interest occur on the tainted data, an operational trace of these operations is generated and written to disk. Once the application finishes processing the input, an offline analysis component plays back the operational trace and builds symbolic expressions for the sinks and branches that it is interested in.

At compile time, a modified LLVM compilation pass, which runs after all optimization passes have run, adds code to keep track of taint and to log information required to generate the operational trace. Taint is tracked using a shadow memory model. Whenever input bytes are read from the outside, be it a file or the network, a label is assigned to those bytes and the corresponding shadow memory bytes marked with it. Instructions added at compile time make sure that taint is propagated as the input bytes are manipulated by the program. A runtime library that is automatically linked into the instrumented program at compilation time, provides wrappers for sinks that are represented by function calls, e.g. `malloc` or `memcpy`. Calls to these functions by the program are replaced by the compilation pass to calls to the instrumentation wrappers. When the wrappers are called, they check if their arguments have taint associated with them, and if so they log all information needed to generate the symbolic expression for that sink. Similarly, for non-function-call sinks such as array index accesses, the compilation pass adds code *before* the instruction to check if the index is tainted and if so, log messages with information to create the array index sink.

An important consideration for the system is to have low runtime overhead by minimizing the amount of data that it generates as the application runs. Instead of logging every operation performed by the application, only certain information which cannot be determined at compile time is logged. At analysis time, this information is coupled with LLVM IR [13] bitcode files [11] that are generated as the program is compiled. The coupling is done by assigning IDs for each instruction executed by the program, and referring to those in the operation trace.

The first scenario that needs to be logged are instructions for which the operands cannot be determined at compile time. For example, loads and stores:

```
%val = load i32, i32* %ptr
%store i32 %val, i32* %ptr
```

Or PHIs [14]:

```
Loop:       ; Infinite loop that counts from 0 on up...
  %indvar = phi i32 [ 0, %LoopHeader ], [ %nextindvar, %Loop ]
  %nextindvar = add i32 %indvar, 1
  br label %Loop
```

The next scenario are function arguments and return values because at compile time we can't determine who a function's caller is. Thus at runtime

we need to log the ids of the arguments and the id of the value being returned:

```
define i32 @foo(i32 \%first , i32 \%second) {
  %val = add i32 %first , %second
  ret %val
}
...
%foo1 = call %i32 @foo(%val1 , %val2)
```

The third scenario is when statements of interest are reached with tainted arguments as mentioned before. In addition to sinks, conditional branches are also logged when the direction that they took depends on tainted data.

The final type of data that needs to be logged are concrete values used by tainted binary operations in case one of the operands is not tainted:

```
1 %val1 = load i32, i32* %ptr1
2 %val2 = load i32, i32* %ptr2
3 # Need concrete values in case one of val1 and val2 are not tainted
4 %new_val = %mul i32 %val1 , %val2
5 store i32 %new_val , i32* %ptr
```

We don't need to log the actual multiplication operation, nor the IDs of their operands, since we can figure out all this information by looking at the bitcode file. What we do need to log are the concrete values in case one of the operands is not tainted.

Here, when we see the logged message for store on line 5, we look at the bitcode to find the expression for `new_val`. We see that it's a `mul` of the two values loaded, `val1` and `val2`. If only `val2` is tainted, then we have no way to get at analysis time the value/expression for `val1`, so we use the concrete value logged by the binop and combine it with the expression we have for `val2`.

Note that in this example, one of `val1` or `val2` must be tainted, otherwise the store would not have been tainted and the binop would not have been tainted.

This 2-step process where at runtime only the minimum amount of work is done to generate an operational trace with only data that is not available at compilation time, and the expensive work of analyzing the trace and generating symbolic expressions is done offline has a number of benefits:

- Lower runtime overhead. As long as writing out the data (to disk or over the network) is optimized, there is a minimum amount of work being done as the application is running.

- Enables more powerful offline analysis of the operational trace, for example, loop summarization, than what would be possible if analysis were to be done as the application is executing.
- The operational trace can be used for other purposes besides the generation of symbolic expressions.

As an example, consider the following C program.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int gbuf[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
5  unsigned char fbuf[25];
6
7  void __attribute__((noinline)) print_idx(unsigned int idx) {
8    int n = gbuf[idx];
9    printf("The number at index %d is %d\n", idx, n);
10 }
11
12 int main(int argc, char **argv)
13 {
14   FILE *f = fopen(argv[1], "rb");
15   fread(fbuf, 1, 25, f);
16   fclose(f);
17
18   print_idx(fbuf[16] + 2);
19
20   return 0;
21 }
```

We're interested in exploiting a vulnerability on the global array access at line 8.

A redacted version of the corresponding LLVM IR follows which we will use to drive the following discussion. Note that each global variable, instruction, and function argument has been assigned an ID, each within its own namespace. These IDs are generated at compilation time, and used to create references from the operational trace back to the bitcode files.

```
1  @gbuf = common local_unnamed_addr global [10 x i32] {id: G0}
2  @fbuf = common global [25 x i32] {id: G1}
3
4  define void @print_idx(i32 %idx {id: A00}) {
5  entry:
6    %idxprom = zext i32 %idx to i64 {id: 0}
```

```
7   %arrayidx = getelementptr @gbuf, i64 0, i64 %idxprom {id: 1}
8   %0 = load i32, i32* %arrayidx {id: 2}
9   %call = tail call i32 (i8*, ...) @printf(...) {id: 3}
10  ret void
11 }
12
13 define i32 @main(i32 %argc {id: A10}, i8** %argv {id: A11}) {
14 entry:
15  %arrayidx = getelementptr %argv, i64 1 {id: 4}
16  %0 = load i8*, i8** %arrayidx {id: 5}
17  %call = tail call %struct._IO_FILE* @fopen(...) {id: 6}
18  %call1 = tail call i64 @fread(...) {id: 7}
19  %call2 = tail call i32 @fclose(...) {id: 8}
20  %1 = load i32, i32* getelementptr @fbuf, i64 0, i64 16) {id: 9}
21  %add = add i32 %1, 2 {id: 10}
22  tail call void @print_idx(i32 %add) {id: 11}
23  ret i32 0
24 }
```

And here is the operational trace from running the program:

```
1  ALLOC: id: G0, address <gbuf>
2  ALLOC: id: G1, address <fbuf>
3  FN_START : main
4  FREAD: offset: 0 size: 25 dest: <fbuf>
5  LOAD: id: 9 : source: untainted — <fbuf + 16>
6  FN_START : dfs$print_idx
7  ARGUMENT: id: A00 num: 0, runtimeID 10
8  LOAD: id: 2: source: tainted
9    ADD SINK FOR TAINTED LOAD WITH ID 2
10 FN_END : dfs$print_idx
11 FN_END : main
```

The trace is processed offline top-to-bottom with the analysis program rebuilding the program's state as it goes, and collecting any necessary information from this state whenever statements of interest are found. However, to make the example easier to follow we will explain things as if the trace was processed from the end.

In order to exploit the vulnerability, we need to generate the following symbolic expressions:

- An expression representing the size of the array that we want to overflow, in this case `gbuf`.
- An expression for how the input is used as an index into the array.

The array index operation happens at the `LOAD` with a tainted source on line 8 of the trace:

```
LOAD: id: 2: source: tainted
```

The ID for the LOAD, 2, is in the trace. We go over to the bitcode file and look up the instruction with ID 2, which we find on line 8 of the bitcode listing:

```
%0 = load i32, i32* %arrayidx {id: 2}
```

The source for the load is tainted, which in LLVM IR translates to the pointer operand used by `load`. We want the symbolic expression for it. Parsing the bitcode tells us that the pointer operand is `%arrayidx` and we find that it is assigned to on line 7 of the bitcode:

```
  %arrayidx = getelementptr @gbuf, i64 0, i64 %idxprom {id: 1}
```

This is a `getelementptr` instruction which is used in LLVM IR to calculate offsets into buffers. This instruction gives us the information we need to generate the first symbolic expression, representing the size of the array that we want to overflow. The size of the array is the size of the global static buffer `gbuf` at line 4 of the source code listing. Parsing the LLVM bitcode file gives us its size as 10 32-bit integers, or `40 bytes`:

```
@gbuf = common local_unnamed_addr global [10 x i32] {id: G0}
```

Back to computing the expression for the tainted pointer operand, the analysis knows how to calculate `getelementptr` expressions from their operands, which leads us to the expression:

```
%arrayidx = <gbuf> + %idxprom * sizeof(i32)
```

`sizeof(i32)` comes from knowing that the size of each element in `gbuf` is a 32-bit integer, as shown on line 1 of the bitcode listing. 32-bit integers are represented in LLVM IR as `i32`.

`%idxprom` is a zero-extension of `%idx` from 32-bits to 64-bits, so the expression is now:

```
%arrayidx = <gbuf> + ((64) %idx) * sizeof(i32)
```

We now lookup `%idx` and find that it is an argument to the current function, `print_idx`. Unlike `%arrayidx` and `%idxprom`, we can't statically figure out an argument's expression, so we go back to the operational trace and get the logged argument with ID A00 (line 7):

```
ARGUMENT: id: A00 num: 0, runtimeID 10
```

The record for this argument contains a `runtimeID` field which represents the ID of the actual argument passed as the program was running. To expand on this, if the `print_idx` function were called 10 times from different parts of the program or with different arguments, the operational trace would have 10 messages each with a different `runtimeID` corresponding to the argument that was passed that time. To know which of the 10 `ARGUMENT` messages corresponds to a subsequent message in the trace that references it, the instrumentation analysis uses the `FN_START` and `FN_END` messages (lines 3, 6, 10, and 11) to create scope within the state that it builds. In this manner, which closely resembles function scopes in programming languages, only one specific `ARGUMENT` is valid at any point in the trace.

Continuing with our example, the `runtimeID` is 10. We look up in the bitcode the instruction with this ID, and find it on line 21:

```
%add = add i32 %1, 2 {id: 10}
```

The analysis knows how to create expressions for adds, and uses the information from the bitcode to continue expanding the expression, which now becomes:

```
%arrayidx = <gbuf> + ((64) %1 + 2) * sizeof(i32)
```

We now follow the operands for the `add` instruction, namely `%1`, which is another load on line 20:

```
%1 = load i32, i32* getelementptr @fbuf, i64 0, i64 16) {id: 9}
```

Unlike the first load, the trace tells us that its pointer operand is untainted:

```
LOAD: id: 9 : source: untainted — <fbuf + 16>
```

We do know that the data it loaded is tainted since it was used as the argument to `print_idx` and eventually as the pointer to the `load` sink that we are trying to exploit. Therefore we now move into getting the expression for the data that was read from memory by the `load`. Back on the trace, the message for the `LOAD` gives the concrete address that was used, in this case `fbuf + 16`.

Part of the state built by the analysis program as it processes the operational trace is a memory map which contains the symbolic expressions and input provenance for each of the tainted bytes in the program's memory.

In our example, we lookup `fbuf + 16` inside of the memory map and find that it was originally tainted by an `fread` call, as shown on line 4 of the trace:

```
FREAD: offset: 0 size: 25 dest: <fbuf>
```

The arguments tell us that 25 bytes were read from the file passed to the program starting at byte 0 of the file. Since `fread` is an input source, we have reached the end of the process and produce the final expression for the tainted pointer operand to the array index sink that we want to exploit:

```
%arrayidx = <gbuf> + ((64) (file_byte_16 + 2)) * sizeof(i32)
```

The resulting symbolic expression is as follows:

```
; Symbolic Variables
(declare-fun **value_0_0x10_0x10 () (_ BitVec 8))
(declare-fun index_expr () (_ BitVec 64))

; Expression
(let ((?x18 (bvmul ((_ zero_extend 32)
                    (bvadd ((_ zero_extend 24) **value_0_0x10_0x10)
                           (_ bv2 32)))
                   (_ bv4 64))))
(let ((?x19 (bvadd (_ bv94839930991040 64)) ?x18)))
(= index_expr ?x19)))
```

As part of the symbolic expression we can see a number of type conversions, first from 8-bit char to 32-bit int, then to a 64-bit integer, and because the target array is made up of integers, we see that the index value given to the array is multiplied by 4 to get to actual byte offsets. The final part of the expression shows the base address of the global buffer, represented as `bv94839930991040`, being added to the computed index value to get to the memory address that we're interested in accessing.

To finish up the example, in order to exploit a vulnerability we need to ask the solver if it is possible to go past the bounds of the static array by setting a particular value in the input file. In this case, the solver would give a solution of 10 (or any value larger than 8) at byte 16 of the input file would cause an overflow.

2.3 Targeted Input Synthesis

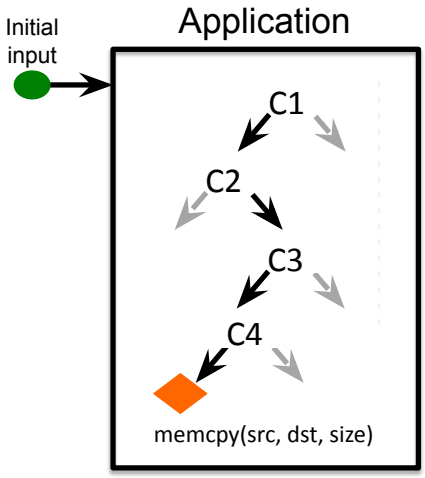
Target input synthesis (TIS) is one of the core techniques used by Aikido's automatic vulnerability injection. First introduced in [24], TIS is a system for automatically generating inputs that trigger vulnerabilities on target applications. At a high-level, given a target statement in a program, for example

a memory allocation site, an array index, a memory transfer operation, or a conditional branch, and an input that reaches that statement, TIS mutates the input to set a desired value at the target statement, for example:

- overflow allocation size,
- array index out of bounds,
- memory transfer size that overflows destination buffer by X amount of bytes
- drive application towards specific code of interest by manipulating branch conditions, etc.

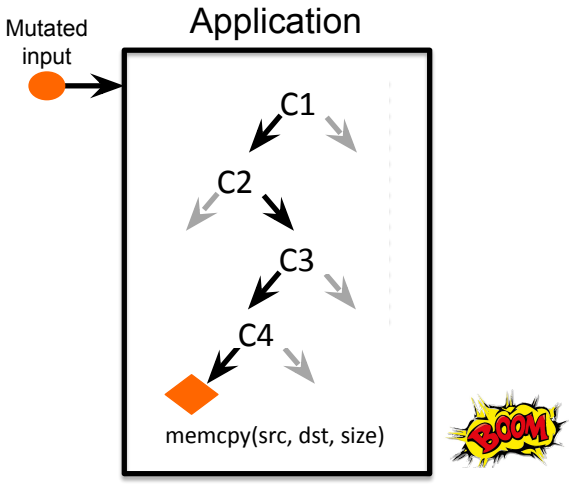
Starting with an input to a target program, TIS instruments the application and identifies statements of interest (*sinks*) and branch operations exercised by the input, and, using an SMT solver, mutates the input to trigger vulnerabilities in those statements of interest.

To understand how TIS works, let us walk through an example of generating an input that triggers a buffer overflow vulnerability at a `memcpy` call in an application. We start by compiling the application to add our instrumentation and running it using a benign input which reaches the `memcpy` call that we want to target:

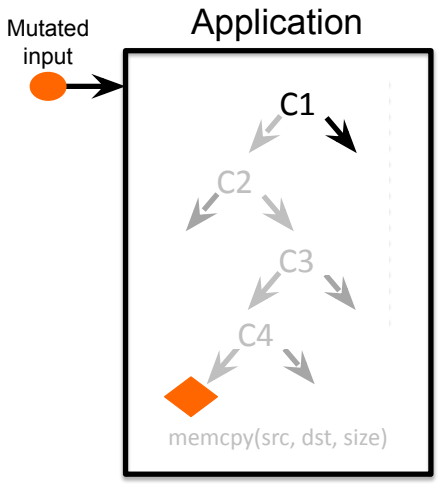


On the initial run, TIS' instrumentation traces the input as it is manipulated by the application until it encounters the target `memcpy` site, the *sink* that we're interested in. Using the trace generated by the instrumentation, TIS builds a symbolic expression tree for each of the parameters to the `memcpy` in terms of the input sent to the application. The inputs that appear in this expression are the *relevant inputs*.

This expression tree is converted into a symbolic equation given to an SMT solver. If the `memcpy` can be overflowed, the solver produces a solution that is used to generate a new *mutated input*. TIS runs the application with the new input. If it detects a crash, it has succeeded in triggering the vulnerability and the process is done.



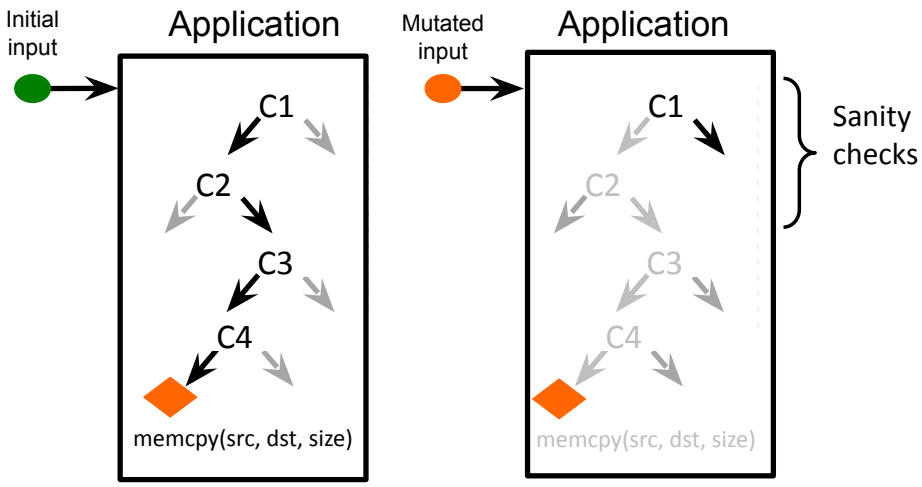
In most cases, however, the mutated input will not cause a crash. Instead, the input will follow a new path through the application which does not lead to our statement of interest.



The cause for this deviation are *sanity checks*, conditionals within the application logic which verify that the input falls within the parameters set by developers. If the input does not pass the sanity checks, the program typically emits an error or warning message and does not further process the input. Failing sanity checks is a common problem with unconstrained SMT solvers which tend to produce extreme inputs. Take, for example, an image processing application, which will have predefined some maximum width and height that it supports, and rejects any input which contains values larger than these.

To trigger an overflow, an input must therefore take the *same path* through the sanity checks as typical inputs that the program processes successfully. One obvious way to traverse the sanity checks is to obtain a new input that 1) satisfies the target constraint to produce the overflow as well as 2) additional constraints that force the solver to generate an input that takes the same path to the target site as the original input which got us to our target sink. This approach ensures that the input passes the sanity checks.

Unfortunately, our results indicate that this approach often fails because, in most cases, the path that the seed input takes through the computation contains additional *blocking checks* that prevent any input that satisfies these checks from triggering the error. To trigger an overflow, an input must take a different path through these blocking checks. The challenge is therefore to find inputs that 1) satisfy the target constraint, 2) satisfy the sanity checks, and 3) find a path through the blocking checks to execute the target site.



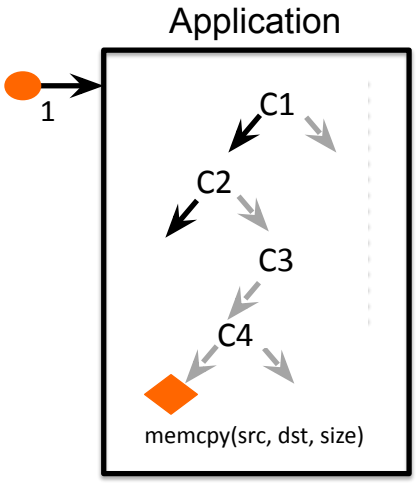
TIS meets this challenge by introducing *Goal-Directed Conditional Branch Enforcement*, a technique that iteratively identifies sanity checks in the application and adds them as constraints to the SMT solver. When TIS detects that no crash occurred with a mutated input, it runs the application again under its instrumentation with the mutated input. It analyzes the instrumentation trace looking for conditional branches whose condition is influenced by the relevant inputs (called *relevant conditional branches*), It compares the path that the seed input followed through the relevant conditionals with the path that the generated input followed. These two paths must differ (otherwise the generated input would have triggered an overflow).

TIS then finds the first (in the program execution order) relevant conditional branch where the two paths diverge (i.e., where the generated input takes a different path than the seed input). We call this conditional branch the *first flipped branch*.

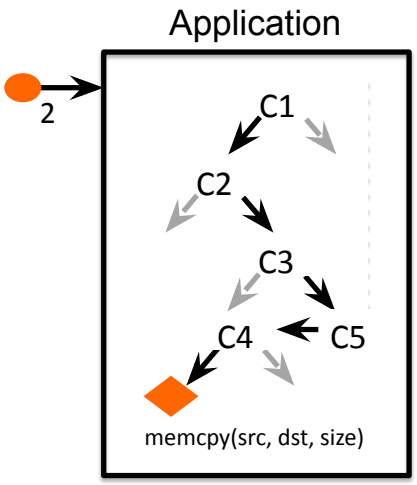
TIS adds the branch constraint from the first flipped branch to the constraint that it passes to the solver, forcing the solver to generate a new input that takes the same path as the seed input at the first flipped branch. In our example, `C1` is the first flipped branch, and the following constraints are passed to the solver:

- Satisfy `C1`
- Overflow `memcpy`

TIS then runs the program on this new generated input to see if it triggers the overflow.



In our example, no overflow was detected. TIS finds that `C2` is now a flipped branch, and adds `not C2` as a new constraint for the solver. A new input `I2` is generated and sent to the application. The `memcpy` is successfully reached and we have an overflow!



Looking carefully at the path taken by `I2`, we can see that it diverges from the path taken by the original input once it reaches `C3`. We can then

say that `C3` was a blocking check that TIS successfully circumvented. As a concrete example of a blocking check, consider the number of iterations for a loop. If TIS were to faithfully enforce that constraint, it would overconstrain the input and fail to trigger the vulnerability.

In general, because the test inputs enforce only relevant branch conditions associated with previously failed relevant sanity checks, this approach gives the input the freedom it needs to navigate the blocking checks that would, if enforced, cause the program to fail to execute the target site (and therefore fail to generate an overflow).

If our example application had not had a vulnerability, TIS would have continued detecting sanity checks and adding them as constraints until the SMT solver returns `unsat` because it is unable to produce a solution that satisfies all constraints in the system.

The following figure summarizes and illustrates the end result of the algorithm in our example:

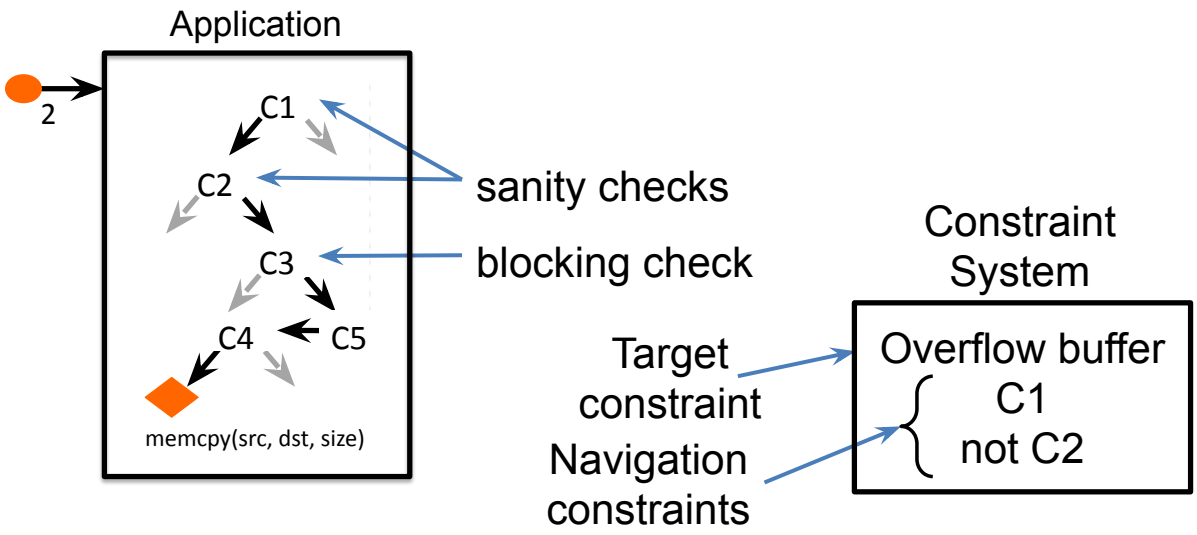
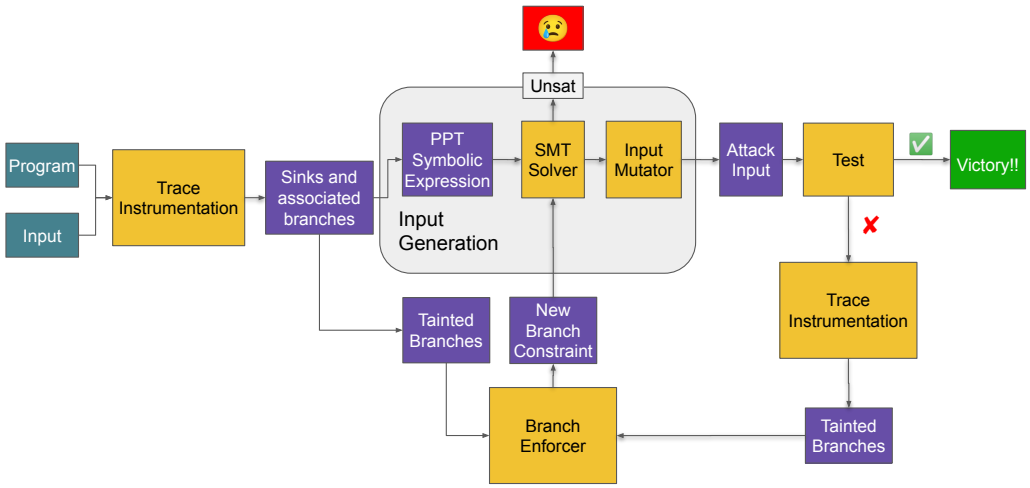


Figure 2.2 shows the general architecture of TIS and its three main components:

- *Trace Instrumentation*: Traces application execution to build symbolic expressions for input values that flow into statements of interest, discussed in Section 2.2.

Figure 2.2: TIS Architecture



- *Input Generator*: Runs the SMT solver with a constraint system that will trigger the vulnerability while safely navigating sanity checks, and modifies the input to match the solution generated by the SMT solver.
- *Branch Enforcer*: Compares execution paths through relevant conditional branches to identify flipped branches which represent sanity checks. Generates new constraints to satisfy these sanity checks and passes them to the Input Generator.

2.4 Comfortfuzz: Orchestration and Automation

We have developed a system, *Comfortfuzz*, that orchestrates the various components of Aikido to inject vulnerabilities.

The system is organized around a set of core concepts that allow it to support many different types of target applications, including those which can be built from source and run locally, and OSS-Fuzz applications which run inside of docker containers. It also supports emulated IoT devices with stripped binaries running inside of docker containers on separate machines, which we have used for other DARPA projects.

The first core concept relates to how to run applications and commands

in the environment where the application is run. Comfortfuzz refers to these as `AppRunners` and they define an API for the rest of the system to set up applications, stage files, execute commands, and manage application lifetime without knowing the details of how these operations occur. Comfortfuzz contains four `AppRunner` implementations:

- *Local*: As the name suggest this does everything on the local machine. This is the simplest implementation, and it was used during Phase I to generate exploits against applications which could be built from source.
- *SSH*: An extension of the `Local AppRunner` for situations where the application needs to be run in a specific environment, due to its software dependencies or any other idiosyncrasies. Under the hood, all operations are done using a passwordless `ssh` connection, `rsync` for transferring files, and `ssh` tunnels for network communication and interaction with the target application.
- *FirmadyneDocker*: Used when targeting emulated IoT devices. The containers run Firmadyne [19] inside, which in turn uses QEMU for the emulation component. It supports having the docker containers on the same machine as where Comfortfuzz runs or on a dedicated remote machine. For the latter case, it leverages Docker's native `ssh` support for container management, `rsync` for transferring files, and `ssh` and `socat` tunnels for communication. In addition, it uses Pexpect [18] to interact with the containers and the emulation. We developed this for use on DARPA's HACCS [8] program.
- *OSS-Fuzz*: Used when targeting applications running under Google's OSS-Fuzz [15] system. It is in some ways similar to `FirmadyneDocker` since OSS-Fuzz uses docker containers under the hood, but the interactions with the docker container are hidden behind a custom API layer that we developed for this purpose. This will be described in more detail later.

Aikido relies heavily on the `Local` and `OSS-Fuzz AppRunners`.

The second core concept is how to manage the various components that make up an application, which Comfortfuzz calls `TestApps` and `AppInstances`. As will be discussed later, depending on the task at hand, Comfortfuzz requires different configurations of the app to be available. A `TestApp` represents the overall application, whereas an `AppInstance` represents an instance

of an app with a specific name and configuration applied to it. In addition to this, these classes provide a virtual filesystem-like (and lite) abstraction that allows Comfortfuzz to make files available locally and wherever the application is run, and also to manage paths to these files so that depending on the needs it can request a local or runtime path and have them point to the same file object.

- *Buildable*: Represent applications that can be locally built from source. It supports compiling and running the application and its dependencies with different configurations.
- *Device*: Represent emulated devices. Most of its methods are no-ops or mock-like to provide a consistent API to the rest of the system. Most of the complexity here lies in making local files available in the emulation docker container and making sure local and runtime paths are kept in sync.
- *OSS-Fuzz*: Represent OSS-Fuzz applications. This perhaps has the most complexity, mixing the ability to compile applications from source in many different configurations, while running them in remote docker containers. It will be discussed in more detail later.

These go hand in hand with the `AppRunners` and interestingly enough their complexity is inversely proportional to the complexity of the corresponding `AppRunner`. `Buildable` apps that are either run locally or through ssh are built from source and are required to support many different compile configurations (e.g. with our DataFlow Sanitizer instrumentation, with Address Sanitizer [2], stock, etc), and the application and required dependencies need to be set up in such a way as to run standalone within the `AppInstance`. On the other hand, `Device` apps are almost mock-like, in that many of their methods are no-ops since they cannot be compiled differently and the whole emulated device is the target application.

The final concept that Comfortfuzz relies on for automation and orchestration is called an `AppInfo` which is a class that provides the application-specific details that are needed to operate on a particular application or device. The base class that the per-application class inherits from determines the type of `AppRunner` and `TestApp` used by the system, with the fields inside of the class filling all the required details. The following classes are available:

- *AppInfo*: Represents a local or remote/ssh standalone application. TIS-generated input will be fed to the application through the command line. An example would be an image processing console application.

- *ServerInfo*: Represents a local or remote/ssh client/server application. The server is the target application that we want to find vulnerabilities in, while the client is a simple, and, most of the time, hand-written, application that knows how to deliver TIS-generated input to the server over the network. An example would be the nginx web server with a simple python HTTP client.

- *DeviceInfo*: Represents a server running inside of an emulated IoT device. Like `ServerInfo`, it has an associated client application that knows how to deliver input over the network.

- *OssFuzzInfo*: Represents an OSS-Fuzz project and the fuzzer application that we're targeting. Like `AppInfo` these are standalone applications that have their input delivered as a file on the command line.

Each target application added to the system has a corresponding `AppInfo` which is created by adding a python module to the `util/app_info` folder with a single function called `get_app_info` which creates the object that will be used for the app. A simplified example of what this looks like for the Jasper application which we used in the vulnerability injection example above follows:

```
def get_app_info(app_args: AppInfoParams) -> AppInfo:
  return AppInfo(
    app_name='jasper-1.900.1',
    exe_name='jasper',
    app_params='-f␣%s␣-T␣jpg␣-F␣/tmp/jasper.jpg',
    env_vars=ld_library_path_env_var,
    soft_timeout=600,
    hard_timeout=660,
  )
```

As can be seen, Jasper is an `AppInfo`, built from source from the application `jasper-1.900.1` and run locally. For this type of application, Aikido has an additional component, a repository where all applications which have been manually set up to be used with the system called `BenchmarkApps`. This repository exposes a standard interface using `makefiles` [6] with standard

targets and environment variables that control the configuration to build. The apps are also built with a standard filesystem layout, with an `install` folder within the `AppInstance`, and corresponding `bin` and `lib` subfolders for the binaries and dependent libraries. The `exe_name` field represents the name of the target binary within the `install/bin` folder just mentioned. `app_params` is a format string which tells Comfortfuzz how to run the application, and more importantly how to pass it input files. In the example above, the `%s` in the format string will be expanded to the path to the input file that the system wants to pass to the application. The `env_vars` field is either a dictionary of environment variables to set when running the application, or a function that should be called to generate such dictionary. In this case, we're passing a standard function that sets `LD_LIBRARY_PATH` to point to the `install/lib` folder in the `AppInstance` to make sure the application is linking with the correct set of libraries. Which is of importance when Aikido injects a vulnerability in a dependent library instead of the main application. Finally the `timeout` fields guarantee that the automation does not block indefinitely when applications misbehave. For example, certain inputs or vulnerability injection modifications may lead the application into an infinite loop.

2.4.1 OSS-Fuzz Applications

The example above represents an application that was onboarded into Aikido by hand, with a human operator understanding the application's build system and manually generating the `Makefile` that Comfortfuzz expects. As mentioned before, this process does not scale for a large number of applications. To address this, Aikido has support for applications which belong to Google's OSS-Fuzz system [15]. OSS-Fuzz is a system for continuous fuzzing for open source software. It aims to make common open source software more secure and stable by combining modern fuzzing techniques with scalable, distributed execution. Open source software opts into OSS-Fuzz by following their guidelines to submit [16] and set up [17] a new project into the system.

To set up a new project into OSS-Fuzz, a project maintainer creates one or more targets to be fuzzed [10], and integrates them with the project's build and test system. A fuzz target is a function that accepts an array of bytes and does something interesting with these bytes using the API under test, for example:

```
int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
```

```
  DoSomethingInterestingWithMyAPI(Data, Size);
  return 0;
}
```

Ideally, all fuzz targets are maintained by the owners of the code, is built with the rest of the code and tests for the project to avoid bit rot, and has a seed corpus with provides good coverage of the part of the application that the target corresponds to.

One of the key aspects of OSS-Fuzz is it forces maintainers to abstract away the details of the application's build system. As mentioned before, many build systems exist, and new ones are created continuously, the less OSS-Fuzz knows about them, the better. To accomplish this goal, OSS-Fuzz requires that for every fuzz target `foo` in the project, there is a build rule that builds `foo_fuzzer`, a binary that:

- Contains the fuzzing entry point.
- Contains `LLVMFuzzerTestOneInput` and all the code it depends on.
- Uses the `main()` function from `$LIB_FUZZING_ENGINE` (an environment variable provided by OSS-Fuzz environment). Since the build system supports changing the compiler and passing extra compiler flags, the build command for `foo_fuzzer` looks similar to this:

```
# Assume the following env vars are set:
# CC, CXX, CFLAGS, CXXFLAGS, LIB_FUZZING_ENGINE
> make_or_whatever_other_command foo_fuzzer
```

Key to this requirement is that there is no point in hardcoding the exact compiler flags in the build system because they a) may change and b) depend on the fuzzing engine and sanitizer being used.

This commoditization of the build system is the key reason we chose to add support in Aikido for OSS-Fuzz. One of the biggest hurdles to onboarding applications into Aikido has been manually understanding how to override the details of the build system, such as the compiler and compiler flags, that are needed during the vulnerability injection process. That problem automatically goes away from the requirements imposed by OSS-Fuzz for all of its applications.

The second key reason for choosing to support OSS-Fuzz is that most of its applications come bundled with a seed corpus. In OSS-Fuzz terms, the

seed corpus is a set of test inputs, stored as individual files, provided to the fuzz target as a starting point (to "seed" the mutations). The ideal corpus is a minimal set of inputs that provides maximal code coverage.

As described in Section 2.3, target input synthesis needs an input that will execute statements of interest. Before supporting OSS-Fuzz, an operator would have to manually find or build inputs for each new application. With OSS-Fuzz, the inputs are automatically found by Aikido since they come bundled as part of the OSS-Fuzz application.

Support for OSS-Fuzz in Aikido covers various aspects:

- Support for the fuzz target entry point (`LLVMFuzzerTestOneInput`) in our LLVM instrumentation
- Provide a replacement `$LIB_FUZZING_ENGINE` library with a `main()` function that fuzz targets can use.
- Provide an API that wraps OSS-Fuzz commands and that Comfortfuzz can use through the `AppRunner`, `TestApps`, and `AppInfo` interfaces.

The first two items are accomplished with modifications to the instrumentation. For supporting the fuzz target entry point, the compiler pass introduces code that wraps the fuzz target's `LLVMFuzzerTestOneInput`. The wrapper function is part of the runtime library and its job is to introduce taint for the data being passed into the fuzz target. In addition, it logs information about the buffer used to pass the data for the offline analysis.

Taint is normally introduced by the instrumentation when file or network data are read in by the program. However, the `main()` function that Aikido provides, which is in charge of reading in the input data and passing it to the fuzz target entry point, is compiled ahead of time (when the rest of Aikido is compiled) without instrumentation. `main()` is compiled into a library that is statically linked automatically by OSS-Fuzz when it builds a particular project. As mentioned above this is accomplished through the `$LIB_FUZZING_ENGINE` environment variable.

OSS-Fuzz Wrapper Interface

Comfortfuzz uses three different abstractions to control OSS-Fuzz applications:

- OSS-Fuzz AppRunner

- OSS-Fuzz TestApp and AppInstance
- OSS-Fuzz AppInfo

OSS-Fuzz provides a command line program, `helper.py` through which applications can be set up, compiled, and run. Applications are encapsulated inside of Docker containers which provide the exact environment and any dependencies that the application needs to compile and run. To bridge the gap between the Comfortfuzz abstractions and the OSS-Fuzz command line program and docker containers, we developed a wrapper interface. The interface is accessed through a command line application, `aarno-cli`, which hides all the details of running `helper.py` and managing the OSS-Fuzz docker containers.

Comfortfuzz assumes that OSS-Fuzz is installed and run on a separate machine and interfaces with the wrapper API through a programmatic `ssh` connection.

There are two top-level commands:

- `aarno-cli image`: Used to build and manage docker images for a particular application. This provides most of the commands required by the `TestApp` and `AppInstance` interfaces.
- `aarno-cli compile`: Used to compile and run applications. This provides most of the commands required by the `AppRunner` interface.

`aarno-cli image` exposes the following subcommands:

- `build`: Build a docker image with a particular ID. The docker image represents a Comfortfuzz `AppInstance`, and the ID matches the ID that Comfortfuzz assigned to the `AppInstance` when it instantiated it.
- `check`: Checks if a docker image with the passed ID already exists.
- `copy`: Makes a copy of a docker image using a new ID. As part of the vulnerability injection process, Aikido copies `AppInstances` many times as it iteratively modifies the code attempting to introduce a vulnerability.
- `read`: Reads a file from the docker image.
- `write`: Writes a file into the docker image.

- `remove`: Removes the docker image from the system.

`aarn-cli compile` exposes the following commands:

- `add`: Compiles the project with the passed image ID using the passed sanitizer. The sanitizer can be Aikido's instrumentation, referred to as `dataflow`, or `address` which refers to compiling the application with Address Sanitizer [2]. The latter is currently used when evaluating whether a particular injected vulnerability makes the program vulnerable.
- `remove`: Counterpoint to `add`, this removes a compilation of the project.
- `check`: Checks if the passed compilation was already done.
- `get-compile-out-path`: Returns the absolute path where files related to the compilation are stored. Comfortfuzz's `AppInstance` uses this to know where to transfer files required for analysis and to transfer files to be given to the application as it runs.
- `get-project-out-path`: Similar to `get-compile-out-path`, returns the absolute path where all the files for a particular project/application are stored. This is used to compute different paths needed to comply with the `TestApp` interface.
- `run`: Runs an application belonging to a compilation with the passed input file inside of its corresponding docker container.
- `stop`: Stops an application that was started by `run`.
- `kill`: Forcefully kills an application that was started by `run`.
- `run-custom`: Runs a custom command inside of the docker container belonging to a compilation. This is used to do instrumentation analysis and perform source code modifications during the vulnerability injection process.

The standardization provided by OSS-Fuzz leads to simple `AppInfo` which require only a minimal small of details before the application can be onboarded into Aikido. In fact, instead of having a file for each OSS-Fuzz application that needs to be targeted, Comfortfuzz is able to consume a *csv* file with 3 columns:

- Project: The name of the OSS-Fuzz project.
- Program: The fuzzer inside of the OSS-Fuzz project to target.
- Corpus: The name of the corresponding zip file containing the input corpus.

From this csv file, Comfortfuzz automatically creates `AppInfo` objects which can be imported as python modules (through a custom importer) and fed into the rest of the system.

2.5 Vulnerability Injection Process

The first step in the vulnerability injection process is identifying statements of interest, *sinks*, to target. Using its instrumentation, Aikido builds symbolic equations for each sink executed by the program in terms of the bytes of the input given to the program. In order to be a viable target, a sink's symbolic equation must meet two criteria:

- If no constraints are present, the target *is* vulnerable.
- With all known constraints present, the target *is not* vulnerable.

The first criteria guarantees that a vulnerability can actually occur at the sink. For example, for an array access sink, if the input that is used as the index of the array is only 8-bits and the buffer backing the array is larger than 256 bytes, then no matter which values are set in the input, the array buffer will never be overflowed.

The second criteria guarantees that the program does not have an existing vulnerability.

The set of viable sinks are found by running the application and corresponding input through TIS (see Section 2.3).

Once all viable sinks and their corresponding constraints have been identified, a symbolic equation for the combined sink and constraints is built. Then, the system iterates over the constraints, removing its corresponding symbolic equation from the overall one, and requesting a solution from the SMT solver. If the solver returns *unsat*, the constraint's equation is added back and the process continues.

If the solver returns a solution, then the system moves onto the next stage. It uses a custom program, `branch-remover` which uses the LLVM Clang API to identify the source code branch that corresponds to the constraint just removed, and "*removes*" it from the source code using the following modification (using our Jasper example from before):

```
if (0 && JAS_CAST(int, sot->tileno) >= dec->numtiles) {
  jas_eprintf("invalid tile number in SOT marker segment\n");
  return -1;
}
```

We found this approach superior to removing the branch and its associated code from the source because it creates very little disturbance in the program. In addition, it means we do not have to deal with situations like orphaned `else` blocks or removing `else if` blocks, which could potentially alter the semantics of the program beyond the removal of the branch.

The modification is done using the OSS-Fuzz API to run `branch-remover` inside the Docker container. This is needed because Clang needs to be able to process the source code for the program in order to analyze it, and in order to do this, it needs to run in the same environment and with the same dependencies that were used at compilation time. This task is aided by a compilation database [3] which is generated at compilation time using Bear [1].

`branch-remover` outputs the modified code to standard output. This is captured by Comfortfuzz, written to a file which is then sent to OSS-Fuzz and written into a new `AppInstance` (backed by its own Docker container). The next step is to empirically verify that the removal of the branch does create a vulnerability.

To do this, the new instance is passed to TIS. If TIS finds an input that causes a vulnerability, we have a candidate branch for removal and can put it in the queue for branch modification. Otherwise, TIS will generate a new set of constraints which still protect the sink from being vulnerable. This can happen in cases where a sanity check was not executed with the original input, but once the branch was removed, a new path was executed which exercised that check.

In that case, the injection process recurses with the new set of constraints, choosing a new constraint to remove at a time as described above. This process can continue until an operator-controlled parameter that sets the maximum number of branches that can be removed is reached, or all protecting constraints have been removed, in which case there is not much left to do.

During this process, care must be taken not to generate duplicate candidates. For example, if initially a sink is protected by branches `A, B, C, D`, and on the first iteration the system finds that removing the set of branches `A, C` leads to a vulnerability, then the system must know that removing the set `C, A` does not represent a new potential candidate.

Once all candidate branches for removal have been found, the system moves to the modification/patching stage.

2.5.1 Conditional Branch Modification

Removing conditional branches is the simplest form of vulnerability injection that Aikido supports. For more realistic vulnerability injection, Aikido performs modifications using a template-based system.

For each template Aikido knows how to create a vulnerability. For example, the system has a template to add an off-by-1 error to a comparison by changing the strictness of the predicate. To match this template, at least one of the operands for the target conditional branch must be tainted, neither one can be a constant, and the predicate is one of `>, <, >=, or <=`.

Using our running Jasper example:

```
if (JAS_CAST(int, sot->tileno) >= dec->numtiles) { ... }
```

This template matches because both `sot->tileno` and `dec->numtiles` are tainted, and the predicate, `>=`, is one of the supported ones. The template then changes the predicate to `>` creating an off-by-1 error.

Given the list of candidates from the removal stage, Aikido iterates over it, trying to match each branch to one of the system templates. Matching happens at both the Clang AST (source code) level and the LLVM IR level.

Matching at the IR level is required to generate new symbolic expressions after the IR has been transformed by the template. The post-transform symbolic expression is passed to the SMT solver to create a proof that a vulnerability exists. Since all instrumentation happens at the LLVM IR level, Aikido has no ability to generate symbolic expressions from source code if it only were to match at the source code level. Matching at the source code level is required because our end goal is to generate source code patches.

The obvious question is whether a mapping can be made between Clang's AST and LLVM IR. We have found this process to be infeasible. To begin with, mapping C source code and Clang's AST is straightforward. There

is a standard one-to-one mapping between the C source and the AST. For example:

```
if (a >= b){
   // Block 1
} else {
   // Block 2
}
```

Generates the following AST:

```
-IfStmt <line:17:3, line:21:3>
 |-<<<NULL>>>
 |-<<<NULL>>>
 |-BinaryOperator <line:17:7, col:12> 'int' '>='
 | |-ImplicitCastExpr <col:7> 'int' <IntegralCast>
 | | `-ImplicitCastExpr <col:7> 'char' <LValueToRValue>
 | |   `-DeclRefExpr <col:7> 'char' lvalue ParmVar 'a' 'char'
 | `-ImplicitCastExpr <col:12> 'int' <IntegralCast>
 |   `-ImplicitCastExpr <col:12> 'char' <LValueToRValue>
 |     `-DeclRefExpr <col:12> 'char' lvalue ParmVar 'b' 'char'
 |- // Block 1
 `-// Block 2
```

And at first glance, it looks like there could be a similar 1-to-1 mapping between the AST and IR:

```
entry:
 %cmp = icmp gte i8 %a, %b
 br i1 %cmp, label %if.then , label %if.else
if.then:
 ; Block 1
if.else:
 ;Block 2
```

However, due to the many optimizations that LLVM may apply to the source code, the straightforward translation can become any number of similar but not identical IR blocks. Here are four additional IR snippets that could be generated from the source code above:

```
entry:
 %cmp = icmp ult i8 %a, %b
 br i1 %cmp, label %if.else , label %if.then
if.then:
```

```
 ; Block 1
if.else:
 ; Block 2
```

```
entry:
 %cmp = icmp ult i8 %a, %b
 br i1 %cmp, label %if.then, label %if.else if.then:
 ; Block 2
if.else:
 ; Block 1
```

```
entry:
 %cmp = icmp gte i8 %b, %a
 br i1 %cmp, label %if.then, label %if.else if.then:
 ; Block 2
if.else:
 ; Block 1
```

```
entry:
 %cmp = icmp ult i8 %b, %a
 br i1 %cmp, label %if.then, label %if.else if.then:
 ; Block 1
if.else:
 ; Block 2
```

And it is impossible to predict which one will be generated, since it depends on the set and order of optimizations applied, and it may easily change as the compiler changes, and optimizations are added, removed, or modified. For this example, the IR block that was actually generated was not the obvious one that we initially showed, instead is the following one:

```
entry:
 %cmp = icmp ult i8 %a, %b
 br i1 %cmp, label %if.else, label %if.then
if.then:
 ; Block 1
if.else:
 ; Block 2
```

Matching from LLVM IR back to C source has just as many problems. Given the IR snippet above there are many possible snippets of C source code that could have generated it:

```
if (a >= b) {
  // Block 1
} else {
```

```
if (b <= a) {
  // Block 1
} else {
```

```
  // Block 2
}

if (!(a < b)) {
  // Block 1
} else {
  // Block 2
}

if (b > a) {
  // Block 2
} else {
  // Block 1
}

if (a+1 > b) {
  // Block 1
} else {
  // Block 2
}

int x = a + 5*b;
int y = b + 12;
int z = x - 5 * y + 72;
if (z >= y) {
  // Block 1
} else {
  // Block 2
}
```

```
  // Block 2
}

if (!(b > a)) {
  // Block 1
} else {
  // Block 2
}

if (a < b) {
  // Block 2
} else {
  // Block 1
}

if (a > b-1) {
  // Block 1
} else {
  // Block 2
}
```

If we consider macros, the are almost no limits to what C source code can generate a particular snippet of LLVM IR code:

```
#define TEST (a >= b)
if (TEST) {
  // Block 1
} else {
  // Block 2
}

#define foo a
#define bar b
#define baz foo
#define qux bar
#define quux foo
#define corge qux
```

```
#define TEST(x, y) (x >= y)
if (TEST(a, b)) {
  // Block 1
} else {
  // Block 2
}
```

```
#define uier quux
int x = uier + 5*corge;
int y = b + 12;
int z = x - 5 * y + 72;
if (z >= y) {
  // Block 1
} else {
  // Block 2
}
```

An important observation about matching at both levels is that the predicate at the source level does not need to match the one at the IR level, so long as they define the same comparison. For example:

```
(%w < $MAX_W)
($MAX_W > %w)
```

Are equivalent even if their predicate is different. Alternately, the two comparisons can be the inverse of each other, as long as their branches are also inverted, for example:

```
(%w < $MAX_W)
(%w >= $MAX_W)
```

Continuing on the modification process, if a match is found, the corresponding transformation is applied. First, the IR transformation is performed, and the symbolic expression for the branch is recomputed. The new symbolic expression is added to the existing system of equations for the sink that we are targetting and passed to the solver. If a solution is found, we have proof of vulnerability. Next, we perform the corresponding source code transformation and recompile the program. Similar to the branch removal process, the source modification process is designed to disturb as little of the source code as possible. To accomplish this, Aikido uses the Clang AST to get the exact start and end of the characters that need to be transformed. Only those characters will be changed. In our Jasper example, the AST returns the bounds for the `>=` predicate that we wish to change, and we use that to change that into `>`. In this manner, untouched macros and other preprocessor directives are left unexpanded. In the Jasper example, `JAS_CAST` is a macro which is left untouched, thus creating a minimal patch for the vulnerability. The IR and AST transformations are built such that no matter the predicates present, their original equivalence is preserved as discussed above.

If all goes well, the modified application is fed back to TIS to try and generate an input that will exercise the vulnerability. If TIS succeeds, Aikido has found a succesful injection along with the symbolic and input proof of vulnerabilities. The final step is to generate a source code patch that represents the vulnerability to be injected.

Chapter 3

RESULTS AND DISCUSSION

We have implemented a prototype of Aikido that runs on Linux. The instrumentation is implemented on top of LLVM version 10. TIS and Comfortfuzz are implemented using Python 3.8. The OSS-Fuzz wrapper supports OSS-Fuzz from August of 2022 (OSS-Fuzz does not have version numbers).

For applications onboarded manually, Aikido is able to inject vulnerabilities on 3 separate programs:

- jasper version 1.900.1:

```
--- orig/jpc_dec.c
+++ patched/jpc_dec.c
@@ -489,7 +489,7 @@
                dec->curtileendoff = 0;
        }

-       if (JAS_CAST(int, sot->tileno) >= dec->numtiles) {
+       if (JAS_CAST(int, sot->tileno) > dec->numtiles) {
                jas_eprintf("invalid tile number in SOT marker segment\n");
                return -1;
        }
```

- dnsmasq version 2.77:

```
--- orig/rfc3315.c
+++ patched/rfc3315.c
@@ -206,7 +206,7 @@
   /* RFC-6939 */
   if ((opt = opt6_find(opts, end, OPTION6_CLIENT_MAC, 3)))
     {
```

```
-       if (opt6_len(opt) - 2 > DHCP_CHADDR_MAX) {
+       if (opt6_len(opt) - 2 > DHCP_CHADDR_MAX + 100) {
          return 0;
        }
        state->mac_type = opt6_uint(opt, 0, 2);
```

- unzip version 60:

```
--- orig/process.c
+++ patched/process.c
@@ -1900,7 +1900,7 @@
          eb_id = makeword(EB_ID + ef_buf);
          eb_len = makeword(EB_LEN + ef_buf);

-         if (eb_len > (ef_len - EB_HEADSIZE)) {
+         if (eb_len > (ef_len - EB_HEADSIZE) + 100) {
            /* discovered some extra field inconsistency! */
            Trace((stderr,
              "getZip64Data: block length %u > rest ef_size %u\n", eb_len,
```

These results show the application of two different transformations. The first converts a non-strict predicate into a strict predicate to create an off-by-1 error. The second adds an arbitrary quantity to a strict check to increase the range of values allowed by the check. Using an arbitrary quantity is useful for scenarios where the buffer being overflowed is part of a larger structure and doing a simple off-by-1 error may not immediately trigger a vulnerability.

3.1 OSS-Fuzz Results

Integrating with OSS-Fuzz opened the door to exploring a myriad of source code bases to apply injections to, allowing us to extensively test the system.

OSS-Fuzz continuously adds new projects. At the time we last synced with its upstream version there were a total of 437 C/C++ projects. From those, we were able to build the base docker image of 346 of them. The failures are due to a number of different reasons, such as incomplete projects, bugs in the corresponding Dockerfile, networking issues trying to download dependencies, and so on.

The next step was to attempt compiling these images with both our instrumentation and address sanitizer. For address sanitizer, we were able to

successfully compile 263 projects, for a 76% success rate. For our instrumentation, we were able to compile 203 projects, for a 59% success rate.

We proceeded to run Aikido on all the fuzzers associated with each of these programs which had an associated input corpus. The system was able to automatically inject vulnerabilities into fuzzers associated with 4 of these applications:

- libsrtp: 2 patches

```
--- orig/fuzzer.c
+++ patched/fuzzer.c
@@ -295,7 +295,7 @@
                                const size_t key_size)
 {
     uint8_t *ret;
-    if (*size < key_size) {
+    if (*size + 100 < key_size) {
         return NULL;
```

```
--- orig/fuzzer.c
+++ patched/fuzzer.c
@@ -458,7 +458,7 @@

     if (params.num_xtn_hdr != 0) {
         const size_t xtn_hdr_size = params.num_xtn_hdr * sizeof(int);
-        if (*size < xtn_hdr_size) {
+        if (*size + 100 < xtn_hdr_size) {
             fuzz_free(policy->key);
```

- gfwx: 1 patch

```
--- orig/datasource.hpp
+++ patched/datasource.hpp
@@ -130,7 +130,7 @@
     (void)id;

     uint32_t getSize;
-    if ( left < sizeof(getSize) ) {
+    if ( left + 100 < sizeof(getSize) ) {
         throw OutOfData();
```

- libexif: 1 patch

```
--- orig/exif-data.c
+++ patched/exif-data.c
@@ -905,7 +905,7 @@
                                 d++;
                                 ds--;
                                 l = (((unsigned int)d[0]) << 8) | d[1];
-                                if (l > ds)
+                                if (l > ds + 100)
                                         return;
```

- lame: 1 patch

```
--- orig/datasource.hpp
+++ patched/datasource.hpp
@@ -130,7 +130,7 @@
     (void)id;

     uint32_t getSize;
-    if ( left < sizeof(getSize) ) {
+    if ( left + 100 < sizeof(getSize) ) {
         throw OutOfData();
```

In terms of failures, we were able to organize them in the following categories:

- *Instrumentation Coverage*: A large number of applications failed due to shortcomings of the instrumentation, in particular, LLVM IR instruction, code constructs, and standard C library coverage. A number of these shortcomings are manifested through the system being able to trace taint throughout the program but being unable to generate a proper symbolic expression for certain parts of it.

- *Macro Support*: For some applications, Aikido was able to identify a conditional branch to remove, but it could not patch it because the conditional was inside of a macro definition, something that `aikido-patch` does not currently support.

- *String Sink Support*: We found many applications with string sinks, calls to the standard C library string manipulation functions to which tainted data flows. Aikido does not currently support injecting vulnerabilities to make these sinks vulnerable.

- *Larger Template Library*: For some applications, Aikido got to the vulnerability injection stage of the pipeline, but was unable to match the target conditional branch to any of its existing templates.

In addition, we found many failures that were due to the way applications are built within OSS-Fuzz. In particular, many applications make use of compile-time generated header files, which the build system stores in ephemeral storage inside of the Docker container. While this works fine for the standard OSS-Fuzz scenario which only needs access to the final executable files, Aikido needs to be able to replicate the compilation steps in order to perform source code modifications. In this respect, while using OSS-Fuzz greatly lowers the bar for onboarding new applications, Aikido has stronger requirements from the build process, and we will need to investigate how to automatically enforce these.

We also found issues with missing debug information. Aikido needs debug information in order to connect LLVM IR information generated by the instrumentation to source level constructs. In the presence of certain compiler optimizations this debug information gets lost. Fortunately, work is ongoing on LLVM's side to preserve as much debug information as possible.

Finally, we found a number of failures related to bugs in the application's build script and Dockerfile, idiosyncrasies such as having functions inside of header files which cannot be processed by the compiler on their own (they're meant to be included by a file which provides the necessary definitions), and mismatch between the Clang version using by Aikido and that expected by the application.

Overall we considered this a great stress test for Aikido and the data collected has provided us with valuable avenues for further development.

Chapter 4

CONCLUSION

We have presented Aikido, a new technique and system for automatically generating and injecting realistic vulnerabilities to real-world applications. Aikido is built to operate on any existing C program, allowing users to create vulnerabilities and evaluate cyber defense security mechanisms on those applications that they are most interested in.

We implemented Aikido as a compiler pass and runtime component in the LLVM Compiler Infrastructure [12], a set of Clang-based tools, and an orchestration and automation system written in Python. Aikido scaled to support hundreds of real open source applications bundled with Google's OSS-Fuzz project and was able to automatically generate and inject vulnerabilities into a number of them.

Inside the Department of Defense (DoD), we envision that Aikido will be used to evaluate the many cyber defense security mechanisms and techniques currently available in the market. In particular, we expect it to be of direct benefit for use by the services within the laboratory environment (e.g., Space and Naval Warfare Systems Center (SSC) Pacific's Combined Test Bed) or a simulated operational environment. We also envision it being used to promote research into more robust security applications, and to provide a standard mechanism for evaluating new mechanisms, for example fuzzers, against existing ones.

In the commercial sector, Aikido will help companies navigate the crowded space of available cyber defense techniques and evaluate the effectiveness of different techniques against their specific code base and particular vulnerability exposure. By better understanding the effectiveness of their cyber defense options the commercial sector can tune their defensive technologies,

tools, and systems to provide an effective defense against their cyber enemies. Commercial benefits include increased cyber warfare protection of critical infrastructure environments (e.g., nuclear, electrical, transportation, etc.).

Bibliography

[1] Build EAR (bear). https://github.com/rizsotto/Bear.

[2] Clang AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.

[3] Clang JSON Compilation Database Format Specification. https://clang.llvm.org/docs/JSONCompilationDatabase.html.

[4] DARPA CGC 2018. Darpa Cyber Grand Challenge (CGC) Binaries. https://github.com/CyberGrandChallenge/.

[5] DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html.

[6] GNU Make. https://www.gnu.org/software/make/manual/html_node/index.html.

[7] Google Fuzzer Test Suite. https://github.com/google/fuzzer-test-suite.

[8] Harnessing Autonomy for Countering Cyberadversary Systems (HACCS). https://www.darpa.mil/program/harnessing-autonomy-for-countering-cyberadversary-Systems.

[9] Juliet C/C++ 1.3. https://samate.nist.gov/SARD/test-suites/112.

[10] LibFuzer: Fuzz Target. https://llvm.org/docs/LibFuzzer.html#fuzz-target.

[11] LLVM Bitcode File Format. https://www.llvm.org/docs/BitCodeFormat.html.

[12] The LLVM compiler infrastructure. http://www.llvm.org/.

[13] LLVM Language Reference Manual. https://www.llvm.org/docs/LangRef.html.

[14] LLVM Language Reference Manual – phi instruction. https://llvm.org/docs/LangRef.html#phi-instruction.

[15] Oss-fuzz. https://google.github.io/oss-fuzz/.

[16] OSS-Fuzz: Accepting New Projects. https://google.github.io/oss-fuzz/getting-started/accepting-new-projects/.

[17] OSS-Fuzz: Setting up a new project. https://google.github.io/oss-fuzz/getting-started/new-project-guide/.

[18] Pexpect. https://pexpect.readthedocs.io/en/stable/.

[19] D. D. Chen, M. Egele, M. Woo, and D. Brumley. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, NDSS 2016, 2016.

[20] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan. Lava: Large-scale automated vulnerability addition. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 110–121, 2016.

[21] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 2123–2138, New York, NY, USA, 2018. Association for Computing Machinery.

[22] S. Sidiroglou, E. Lahtinen, A. Eden, F. Long, and M. Rinard. CodeCarbonCopy. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2017.

[23] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. 2015.

[24] S. Sidiroglou-Douskos, E. Lahtinen, N. Rittenhouse, P. Piselli, F. Long, D. Kim, and M. Rinard. Targeted automatic integer overflow discovery using goal-directed conditional branch enforcement. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 473–486. ACM, 2015.