Killerbeez: Fuzzing Framework to Bring Together the State of the Art

Adam Nichols, Ian Bridges, Benjamin Lipton, Jeff Stewart, Tomas Tillery
GRIMM
{adam,ian,ben,jeffball,tomas}@grimm-co.com

Published: 18 OCT 2019

Abstract: The trend of people increasingly relying on software has continued for several decades and shows no sign of abating. Businesses rely on Windows and the applications which run on it, servers are typically some type of UNIX system, and Apple computers are gaining popularity as of late. The desire for these systems to be stable and resilient to attacks drives the need to find software errors which may compromise them. Many improvements have been made in the field of software testing, with one of the popular ones being fuzz testing, or fuzzing for short. Unfortunately, the implementation details make it difficult to compare or combine different methods, while others are only available for specific operating systems, or limited to cases where source code is available. Killerbeez intends to pull these technologies together and allow them to interoperate. It is scalable, supports multiple operating systems, is extensible, and will have support for testing both kernel and user space applications. Killerbeez's goal is to measure the effectiveness of various fuzzing techniques in a variety of situations so the optimal solution can be applied.

1 Introduction

Over the past few years, coverage-guided fuzzing has become a popular way to find software and hardware vulnerabilities due to advances in publicly available tools such as American Fuzzy Lop (AFL)[1] and its derivatives.[2] Many fuzzers have "trophy cases" consisting of a list of bugs known to be found with that tool to demonstrate their effectiveness against real-world applications.[60][42][1] However, most tools are primarily focused on finding bugs in open source, command line Linux software which reads input from files or standard input. While there has been some exploration to get AFL working on other operating systems[3, 4] as well as supporting network input,[5, 6] these features are often omitted.

Most of the published improvements over the original version of AFL are implemented as forks of AFL.[9, 10, 11, 12, 13, 14] Thus the enhancements are mutually exclusive, short of embarking on a development effort to review the modifications and merge the forks back together, manually resolving any conflicts and incompatibilities. The issue of incompatibility is not limited to projects which are modified versions of AFL. Mixing and matching features from various tools requires a significant amount of effort. It involves setting up a build environment, which in itself can be a challenge, as well as merging together different code bases. Often times the code bases will be written in different programming languages, which means they need to be integrated in some way. Using a tool against a type of software it has never been used against before, such as using a Linux kernel fuzzer against another operating system, can find a large number of bugs[44]. However, in practice, security professionals rarely have the amount of time to invest to get the tools working together, or working in other contexts.

We present Killerbeez, a fuzzing framework which brings together many of the various security analysis tools so they can be used together. Killerbeez supports multiple operating systems, can handle target applications with or without source code, and software with a Graphical User Interface (GUI). Furthermore, the input mutation algorithms, instrumentation, seed selection algorithms, and methods for feeding input data to the target are all easily interchangeable via modular components. This modularity enables two properties (1) easily mixing and matching tactics from different researchers and (2) implementing new algorithms easily. Finally, Killerbeez is scalable, using Berkeley Open Infrastructure for Network Computing (BOINC)[7] to distribute work to multiple nodes from a central server.

Our contributions include:

1. An Application Programming Interface (API) combining the different components of a fuzzer in a pluggable (modular) way to allow for extensibility
2. A collection of existing fuzzers modified to use the API
3. A method for automatically determining which libraries are likely to cause a crash, so those can be targeted while fuzzing
4. A technique for quickly utilizing Intel Processor Trace (IPT) trace information to identify unique code traces while fuzzing
5. Ability to automatically filter out trace data related to non-deterministic code

Section 2 covers background information, section 3 pro-

vides an overview, the implementation is covered in section 4, and sections 5, 6 and 7 cover related work, future, work and conclusion, respectively.

2 Background

There are a huge number of fuzzing tools[1, 3, 4, 8, 15, 20, 21, 25, 24, 29, 30, 32, 42, 47, 48, 49] which are publicly available, many of which are very useful on real-world binaries. However, each tool typically only handles one very specific use case or contains other real-world limitations, and most are not designed to scale out. For instance, there are a number of fuzzers targeting kernel system calls[15, 16, 47, 17], others for fuzzing Input/Output Controls (IOCTLs)[18, 19], and others for fuzzing userland Linux targets that only effectively work with command line programs written in C-like languages[1]. Some only function with source code, only work on 32-bit Linux[48], or require users to manually specify the data format which the target is expecting.[41, 49] Finally, there is a category of tools which work amazingly well on tiny example programs but do not work on production software due to bugs or lack of support for features such as multithreading.[50, 51]

Lack of compatibility with production software is typically not viewed as an issue in academic work, as the problem to address can be scoped based on the tools that are available; one may assume that the problem can be solved in other situations, but leave the proof for future work. The researchers are typically correct in their assumption, however practitioners need tools that work in practice, not theoretical solutions.

In industry, the situation is dictated by the target software, and there is often not a choice as to the implementation's programming language, whether source code is available, what operating system it runs on, or which CPU architectures it supports. This leaves the security professional to choose between the limited set of tools that can handle the specific requirements of their target, many of which will turn out to be mutually exclusive. This lack of available tools also creates a new problem, as a common response to this dilemma is to put together a custom tool which meets their needs, and to do so in the shortest amount of time possible. Furthermore, this results in the same code being re-written for different platforms, or sometimes for the same platform, simply because the security professional was unaware of existing implementations. The adapted tool will likely contain some of the same bugs and limitations that were in the initial version of the existing tool, which may or may not get fixed before it is abandoned.

In short, while the state of security research is advancing rapidly, the tools to bring their benefits to life are sorely lacking. Though there are some fuzzing projects which come close, such as OSS-Fuzz,[20] there are not any fuzzing tools which are freely available, work on closed source applications, are easily extendable, can be run in a distributed manner, and run against Windows, Linux, and macOS applications.

3 Killerbeez Overview

The core components of Killerbeez can be split into two logical categories of orchestration and handling interactions with the target[1] program. The former refers to decisions such as what inputs to use as seed data,[2] which mutation algorithms to use, how to minimize the input corpus and other decisions which are best left to a central controller. The latter category contains actions such as launching the target; feeding the target input data; tracking code coverage; determining when the target is done processing the input; and reporting whether the target crashed, froze due to something like an infinite loop, or executed new portions of code.

3.1 Orchestration

Killerbeez coordinates the entire distributed fuzzing campaign. The orchestration tasks are handled by the Killerbeez "manager," which runs on a central server. After some initial configuration to specify targets and strategies, the manager decides what jobs to schedule next. It tracks targets available for fuzzing, selects which tools to use and how to configure them, manages the corpus of inputs by removing less interesting ones, and dispatches jobs to worker nodes to be executed. It also provides a Representational State Transfer (REST) Application Programming Interface (API) that allows a researcher to trigger actions and extract results manually, or to integrate an external system that does so autonomously. The components of the manager are depicted in Figure 1.

3.1.1 Work Distribution

The most basic role of the manager is to provide an interface for queuing tasks to be executed on worker nodes and processing the results. A BOINC server is used to transmit the work to nodes and receive results. The manager provides a layer on top of BOINC that understands Killerbeez-specific parameters such as the mutator and instrumentation to use, making it simple to submit jobs to BOINC that run the fuzzer with an appropriate command line. Jobs submitted via the manager also automatically set up Killerbeez and the target software, so the worker nodes are not required to have any special software installed besides the off-the-shelf BOINC client.

When jobs complete, the manager uses the BOINC "assimilator" interface to collect the results and update the manager's database. The information inserted includes not only the direct output of the fuzzer (new inputs that

[1] Software under test is referred to as the "target."

[2] Initial inputs which will be modified are referred to as "seeds," and the set of initial inputs used is referred to as the "seed corpus."

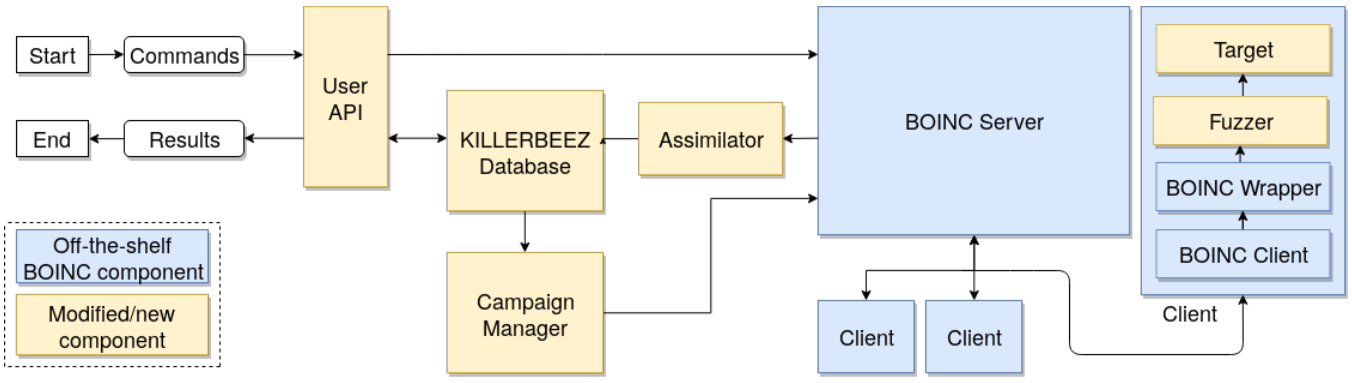

Figure 1: Killerbeez Server Architecture

cause new paths in the binary to be hit) but metadata about the job as well. This metadata could include the average execution time of the binary (to help choose parameters that execute faster), the final instrumentation state from fuzzing (to help the next job find fewer duplicate paths), and various other statistics.

Because the manager is not responsible for running the target application, it does not need to run on the same platform as the target. Thus, it can run on Linux while serving work to be executed by Windows or macOS machines.

3.1.2 Integration

The manager provides a REST API, which allows clients to access and configure seed data, fuzzing targets, and low-level metadata produced during fuzzing. This will enable future enhancements to be made by taking advantage of external tools, such as using a test case generator to produce new seed data. Another planned integration is to use Driller[21] to generate program inputs which reach code that has not yet been reached by mutation.

The REST API is also used for some of the manager's built-in functionality. The campaign manager, the component that plans new jobs to execute, gathers data using the REST API, analyzes the data to determine the next job to create, and then submits the resulting job via the REST API. The corpus minimization uses the REST API to obtain execution traces and modify the working set of seed values. Accessing data via the REST API allows these components to be less coupled with the internals of the manager, enabling them to run as standalone processes. Figure 2 shows how the REST API enables integration with various tools.

3.1.3 Tracing and Corpus Minimization

Killerbeez also introduces the idea of obtaining detailed code coverage information about execution for each input which has a unique code path. This is typically not done by other fuzzers, as obtaining a full execution trace is significantly more overhead than the lightweight instrumentation that AFL or Honggfuzz[42] use.[14] During normal fuzzing, Killerbeez will generally use lightweight methods of tracking execution. However, having a full trace is useful when minimizing the seed corpus and determining which seeds should be weighted more heavily. Reducing the number of files in the corpus helps fuzzers to more efficiently test targets by eliminating inputs which result in the same target functionality being tested.[52, 53] This concept has been encapsulated in the tracer module.

Each time an input is found which hits a new code path, a tracer job can be added via the manager. The new tracer job will be executed by a BOINC client, just like any other fuzzing job. The results will include full trace data, which will be stored in the manager's database. The data can be retrieved via the REST API, enabling a "corpus minimizer" tool to explore the paths covered by the current input corpus and remove inputs that are redundant. More information about the tracer and corpus minimizer can be found in section 4.4.

3.1.4 Work Generation

The manager is also responsible for deciding what work should be performed next. The component that does this is called the "campaign manager," and it consists of several pluggable modules that work together to generate jobs. For jobs that run the Killerbeez fuzzer, the seed selector module specifies an algorithm for choosing the most interesting input to use as a starting point for fuzzing, while the job parameter selector module determines parameters like the mutator to use for the job or the instrumentation options. It is also possible to integrate tools besides the fuzzer into the job system, such as Driller or the tracer. The job type selector module is responsible for choosing which of these tools is currently needed most. The modules can use the REST API to query any of the metadata recorded in the database to make their decisions.

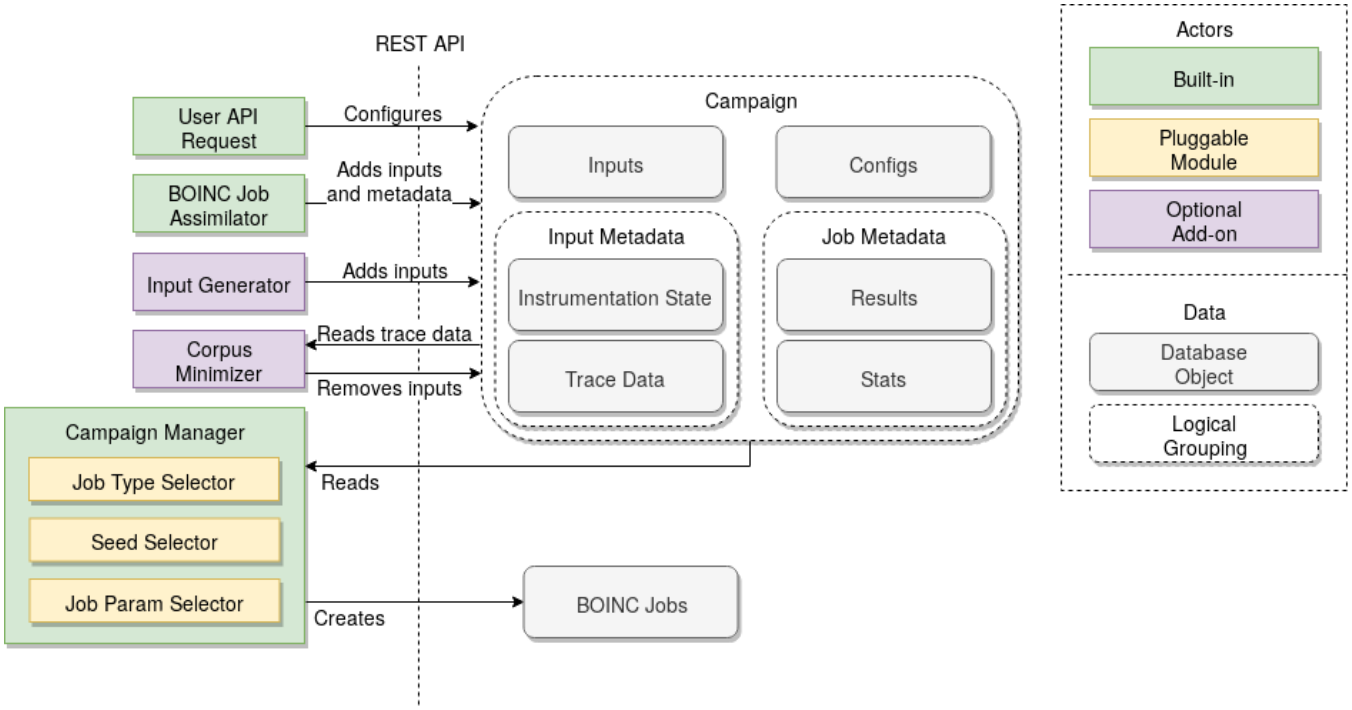

Figure 2: Killerbeez Integration with External Tools

3.1.5 Scalability

The manager can choose the amount of work done per job by specifying things like the number of fuzz iterations. By scheduling larger jobs, the manager can scale up to a very large number of clients, allowing a substantial amount of work to get done with a minimal amount of coordination and network overhead. However, if the manager becomes a bottleneck due to the number of workers or the number of API requests, several components could be scaled out. The BOINC server could be moved to its own machine, or even scaled out to a cluster.[22] The manager database could also be moved to its own machine. The REST API server is stateless, so it could be scaled fairly easily to run on multiple machines, and components that interact via the REST API, such as input generators and the campaign manager, can also be moved to run on their own hardware.

If performance is still a problem even with all components scaled up as much as possible, multiple manager servers could be set up to run parallel fuzzing campaigns, using the REST API to share results between them. In this mode, each manager would act as an input generator for the others.

3.2 Preparation

There is a preparation step in fuzzing workflows which is often overlooked or dismissed, that includes setting up the target software and deciding specific fuzzing parameters. This occurs before any real fuzzing begins and includes things such as compiling the target, possibly with a specific compiler or compiler flags, determining what options should be enabled in the target software, deciding which portions of the code should be tracked for code coverage, and specifying how to deal with non-deterministic code.

The compilation step is driven by the type of instrumentation chosen, which is typically just a matter of selecting the option with the lowest performance penalty. Instrumentation does not need to be added at compile time, but it is added here when possible, as it reduces the overhead at runtime. What options to enable in the target software is target specific and subjective. It depends on the higher level goals. If the goal is to find a vulnerability with wide applicability, choose default options. If it is to test out a specific feature, disable everything except that option. If it is just to find any bug in any configuration, enable everything. While these are important questions, the most interesting decision is what to track in terms of code coverage, and how to deal with non-deterministic code.

The solution AFL uses is to require the user to manually determine which code in the target to instrument. AFL requires the user to compile the target library or executable with a special instrumenting compiler. Alternatively, AFL can use a modified version of QEMU while fuzzing to instrument all of the libraries used by a target at run-time. As expected, instrumenting all of the libraries has a higher overhead than only instrumenting a few specific modules. Killerbeez improves on this by including a tool called the “Picker” which automatically determines which libraries should be instrumented. The algorithm for doing so is described in section 4.5.

One important detail is that the picker can not operate effectively when the target does not have deterministic execution. If feeding the same file into the application multiple times results in different code being executed, this is a problem, not only for the Picker, but also for code coverage in general. Non-deterministic code causes new code paths to be taken, making it appear as if the input file was the reason new code was executed. This results in erroneously keeping inputs which are not actually valuable. AFL does not deal with this problem directly, but it does alert the user to the fact that the target is non-deterministic. The user can then do things like hijack calls to functions such as `srand()` which intentionally introduces randomness and non-determinism. This is often done for applications which employ cryptography for initialization vectors and nonces. Under command line Linux applications, hijacking non-determinism introducing functions works fairly well. On GUI applications in Windows, it does not work as well. There are system calls in Windows which occasionally fail for no apparent reason. This should not be a problem for the target software, as it should be checking the return code to detect and handle this appropriately. However, when these failures happens it has the side effect of making the fuzzer misinterpret the new code coverage to think an uninteresting input was interesting. The details of how nondeterminism is handled in Killerbeez are also described in section 4.5.

3.3 Execution

Execution is handled by the client fuzzer program, which is aptly named "fuzzer." This can be run manually from the command line, however it is typically run by the manager, via a BOINC client. In either case, the fuzzer is responsible for running the target, feeding it input data, tracking code coverage, detecting crashes, and dealing with user interaction such as dialog boxes.

The fuzzer consists of glue code that combines together various modules, which is where all the interesting things occur. The purpose of the Driver, Mutator and Instrumentation modules used in the fuzzer are covered in sections 3.3.1, 3.3.2, and 3.3.3, respectively. The relationships between these components are depicted in Figure 3. The modules which currently exist and how they work are covered in the Implementation section.

The same code base is used on Windows, Linux, and macOS to enable as much code re-use as possible. Most of the mutators are shared among all platforms. Only the Radamsa mutator, which runs as a separate process, has platform specific code. Some of the instrumentation and driver[3] modules, such as the Intel Processor Trace (IPT) instrumentation and WMP driver, contain platform specific and sometimes target specific code.

[3] "driver" refers to driver modules, not operating system drivers.

3.3.1 Driver

Killerbeez offers drivers, which are target-specific wrappers that abstract away the concept of loading data into a target and enable finer definition of the failure modes of a particular piece of software. While typical fuzzers look for crashes and hangs, specifically-written drivers can have more context about a given fuzz target. Better understanding of the fuzz target's behavior means that Killerbeez can make better-informed decisions about the status of a target after a particular input, and it can terminate or classify the result of a particular input more quickly than waiting for a timeout.

First, the driver module is responsible for feeding inputs to a target. This is a departure from most fuzzers, which only work for one type of input. In the case of AFL, the input is a file (or standard input (stdin), which is also a file under UNIX). Syzkaller[15], on the other hand, uses only system calls. Each tool then has to implement their own mutation algorithms, code coverage, results collection and so forth. Drivers enable Killerbeez to reuse all of these components and select how to interact with the target by simply selecting the appropriate driver. Abstracting this away allows for more exotic use cases, such as fuzzing IOCTLs, network servers, network clients, Interprocess Comunication (IPC) such as Mach Messages, XPC, Distributed Objects, Common Object Model (COM), and others, all with minimal effort.

The second thing drivers are responsible for is dealing with any target specific issues, such as handling GUI interactions. For example, if a PDF file with a malformed header is given to a PDF reader application, it typically will pop up a dialog box indicating that the file is corrupt. Fuzzers such as Honggfuzz or WinAFL[4] would wait until a timeout expires.[4] This results in all executions which hit this code path to appear as a hung process. In Killerbeez, a driver could be written for the specific PDF reader which monitors the application for dialog boxes, detects when dialog boxes appear, analyzes the text of the dialog box and determines that the status is a clean exit rather than a hang. This would allow the fuzzer to move on to the next input more quickly, as it would know the input is done being processed and would not need to wait for the full timeout period. It also helps discern between a hang, which may indicate a denial of service vulnerability such as an infinite loop, and an error which is handled in the expected manner. For another example, see the Windows Media Player driver in section 4.1.

Many of the drivers work on many fuzzing targets in a particular category. Targets which accept input from the network are handled by the Network Server driver module, programs which open a file are generally handled by the file driver and so forth. Other drivers can be written to handle things which are specific to particular pieces of software. Some targets will handle opening files differ-

[4] WinAFL can exit at the end of a function, but dialog boxes tend to prevent that function from returning in practice.

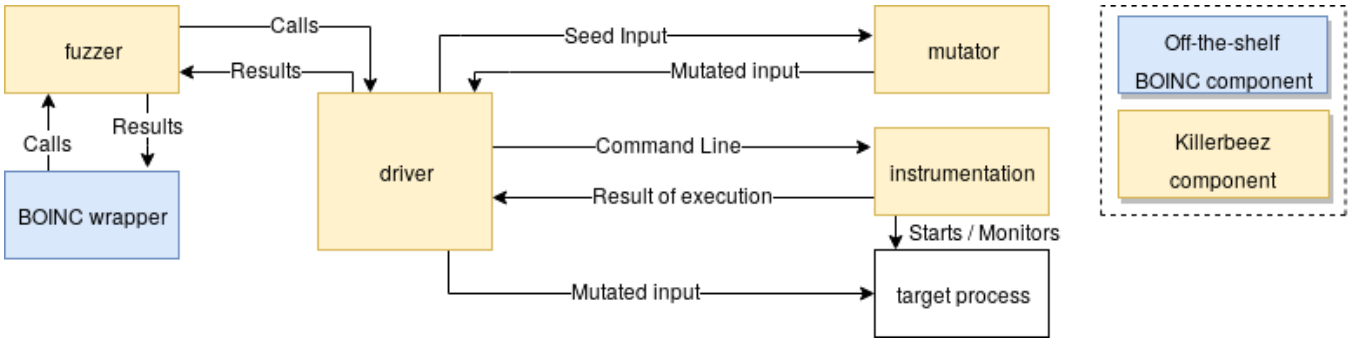

Figure 3: Killerbeez Fuzzer Overview

ently if opened via double clicking an icon as compared to using the open option from the file menu. Other examples include error message analysis to determine if the system should move on to the next input, or if it should click "OK" and continue (e.g. in the event of a warning message).

3.3.2 Mutator

Killerbeez also implements "mutators," which are abstractions on modifying program input. They decide where to modify bytes in the input data, and how to modify them.

Killerbeez uses a selection of user-selectable mutators. Parameters are passed to the mutator module via the driver, which control the operation of the mutator. For example, the bit flip mutator flips a parametrized number of bits throughout the entire input, one at a time.

Modular mutators also enable trivial combination of different approaches. Using the multipart mutator, different mutators can be applied to different parts of the input. This is required for efficiently fuzzing network protocols, as it is often desirable to not mutate the initial packets as they may include a handshake or authentication. Any mutation to this section would prevent the majority of code from being executed, as the target software would execute an error path instead. It can also be used to ensure that the first few bytes in a file are not modified so the file will still be recognized as being the correct file type.

3.3.3 Instrumentation

Instrumentation modules are responsible for tracking program execution and determining if an input has caused the target program to execute new code. How it does this, and what level of granularity is used, are questions left to the module author. There is an IPT instrumentation module which is very high resolution. If a loop somewhere in the target is executed 178 times instead of 177 times, it will detect this as a new code path, as the state explored is different than what was seen before. The AFL instrumentation module, on the other hand, would not consider this to be an input which causes the execution of new code. The AFL instrumentation module uses a bucketing system that groups executions of the same code and considers anything which executes a portion of code 128-255 times to be equivalent.[54]

Sometimes instrumentation modules need to interface with kernel drivers, which are implemented differently on different operating systems. For instance, the IPT instrumentation module uses the perf subsystem on Linux, which is not available on macOS or Windows. Other instrumentation technologies, such as Intel's Pin [23], have a very similar interface across different operating systems, which means more of the code in the instrumentation module can be re-used, simply using #ifdef directives if there are portions which are specific to a particular operating system.

4 Implementation

Killerbeez is an interoperable fuzzing architecture that is environment- and platform-independent. It consists of driver, instrumentation, and mutator modules on the client side, a tracer which obtains accurate code coverage and a picker which determines which code should be instrumented. Scalability is achieved by using a BOINC server to allow multiple client nodes to fuzz in parallel. Clients simply obtain work from and return results to the server.

While there are some novel improvements in Killerbeez, the primary benefit is getting many existing tools to work together and run on new platforms. Each tool pulled into Killerbeez was the best in class on its own, but when combined with others, the value is more than the sum of its parts.

4.1 Driver

Driver modules use a mutator module to mutate an input, an instrumentation module to trace the target's execution, and are responsible for getting the input data into the target. A simple example is a file-based driver, which will create a file containing the mutated input and use

the instrumentation module to launch the application in a way that it will read this file. This is typically accomplished by passing a filename on the command line. For targets which do not have any way to specify the input on the command line, a custom driver would be required to use keyboard shortcuts, mouse input, or some other method of getting the input into the program.

The following drivers have been implemented:

- **File** - for programs that read input from a file
- **Stdin** - for programs that read input from standard input
- **Network Server** - enables fuzzing server programs
- **Network Client** - enables fuzzing client programs
- **Windows Media Player** - for Windows Media Player

The File and Stdin drivers provide feature parity with AFL, in terms of input methods. There is nothing particularly novel about these drivers, but they are an important feature. Sending malicious files via email is a popular attack vector, so there is an interest in proactively finding bugs that can be triggered by loading files.

The Network Server and Network Client drivers provide feature parity with AFL when it is combined with Preeny,[6] which modifies the behavior of network-based target programs to accept input via stdin. Two drivers are needed, one to establish a connection to a server, and the other to accept a connection from a client. These drivers caused the creation of the multipart mutator, which is covered in more detail in Section 4.2.

A Windows Media Player (WMP) driver was created to demonstrate how to deal with a GUI application which does not exit without user interaction. Applications with a GUI are difficult to fuzz and many fuzzers[1, 42, 41, 48, 49] evade this problem by not supporting targets with a GUI. The typical recommendation is to fuzz the library which does the heavy lifting, or modify the application to not load a GUI. This does not work well when dealing with closed source applications. A test harness can be written which calls the undocumented functions in the closed source library, but there is no guarantee that bugs found will also be present and reachable in the real application. This is due to constraints which may be placed on function arguments in the main application, or that functions are called in a different order. While writing a custom harness to test a library is a good recommendation, it should not be the only option. Killerbeez addresses the problem of fuzzing GUI applications with modular drivers instead of simply avoiding the problem.

The next issue a driver has to deal with is determining when the target is done processing the input. Typically, a fuzzer mutates the input, feeds it to a target, and then monitors the target for interesting behavior such as a crash or a hang, and if it does not exit after the timeout period, it considers the target to be hung. This works for command line programs which exit immediately after processing the input, but falls apart when dealing with GUI applications. Setting a timeout which is too low results in stopping before the program is finished processing the input, while setting it too high means wasting time. On top of this, every non-crashing test case is considered a hang. WinAFL attempts to address these problems by forcing the user to reverse engineer the target software to identify the function which reads and processes input data. This is time consuming, and if there is one function which reads the data into memory and another function which parses it, this strategy does not work well. This can sometimes be worked around by going up a level in the call stack until the function is located which calls both the reading and parsing functions. However, that function might also be the one which loads the GUI, which means it may never return. This breaks WinAFL's assumption that there is a function which reads input data, parses it, and returns, which means it will fail in the same way fuzzers intended for command line programs will fail: with every test case being a hang. The next alternative is to patch the target executable to exit after parsing. All of these approaches require reverse engineering, and will have varied results depending on the details of the target software.

The WMP driver, when used with the DynamoRIO[46] instrumentation, uses the same strategy as WinAFL, where a specific function name or offset needs to be specified and the test is ended when that function returns, however this is not the only stopping condition. The driver also checks for sound playing and assumes that if it was able to decode the file and start playing sound, that the application has finished parsing the input file. The underlying assumptions are that errors will be in the code which does the parsing, rather than the code which does the playing, and that all the parsing is done up front. This does mean that bugs which require a significant number of frames will not be found, as the test will conclude early and kill the application. This is a conscious tradeoff which was made to speed up the number of executions per second by terminating much earlier than waiting for the entire clip to play or the timeout to occur. While this technique is based on fuzzing Windows Media Player, it should work on any media playing application.

4.2 Mutator

The mutators from Honggfuzz, Radamsa, AFL, and Ni[24] are leveraged by wrapping the code from these projects to conform to the Killerbeez mutator API. By defining an API for the mutators, researchers can modify other fuzzers to conform to the Killerbeez mutator API and easily swap in the new mutators.

The following mutators have been implemented:

- **arithmetic** - 32-bit arithmetics, both endians. From AFL
- **bit flip** - Flips various number of bits (1-32). From AFL

- **dictionary** - Inserts or replaces values from a dictionary. From AFL
- **havoc** - Runs multiple mutations on a single input. From AFL
- **interesting value** - Inserts values which are more likely to trigger integer overflows or off-by-one errors. From AFL
- **splice** - Splices two input files together. From AFL
- **afl** - All of the AFL mutators, run in the same manner as is done in AFL
- **honggfuzz** - Mutation algorithm from Honggfuzz[42]
- **multipart** - Input must be made up of multiple parts, different mutators are applied to different parts of the input. Useful for network protocols where there is a desire to not disrupt the handshake/login
- **ni** - Mutation algorithm from Ni
- **nop** - Mutator which does not mutate anything, useful for testing and when combined with the multipart mutator
- **radamsa** - Mutator which wraps the Radamsa[25] executable
- **zzuf** - Mutation algorithm from zzuf[29]

As indicated in the list above, several of the mutators were taken from AFL and adapted to the Killerbeez mutator API. These have proven to be effective algorithms and were a solid starting point.

Honggfuzz has a different set of mutators, some of which are similar to those from AFL, such as Honggfuzz's magic value mutator and AFL's interesting value mutator, however there are slight variations which work better against some targets than others. Honggfuzz is the only fuzzer to have found a critical vulnerability in OpenSSL to date,[42] so clearly it does something different than the other fuzzers. Again, the approach was one of pragmatism: taking the existing techniques and bringing them to new environments, such as Windows with code coverage capabilities.

The multipart mutator's development was driven by the network drivers and the desire to have the handshake or authentication section of the input not be mutated, as this would prevent much of the target's code from being reached. The input is divided into parts and a mutator is run on each part. For any parts which should not be modified, the "nop" mutator, which is described below, is selected. This allows different mutators to be used on different segments of the input and the ones which perform the best can be selected more often by the campaign manager. The multipart mutator could also be used with a file-based driver which is multipart aware, allowing different segments of input files to be defined. This would enable things such as ensuring a file's magic bytes are never modified by using the "nop" mutator on the first segment.

Aki Helin, the author of Radamsa, also wrote a mutation algorithm called Ni. This code was adopted with minimal changes to provide more diversity in mutation methods, and based on Aki's reputation for having novel ideas on how to mutate inputs.

During development, it quickly became clear that having a mutator which does not do any mutation would be handy when debugging issues. This is how the nop mutator was born. It was later used when the multipart mutator was developed.

Radamsa is a general purpose fuzzer, written in Lisp, which came from the Oulu University Secure Programming Group (OUSPG) Protos Genome Project.[45] It works well against a variety of network protocols and file formats, and has found dozens of vulnerabilities.[26] The radamsa mutator module in Killerbeez is a simple wrapper which feeds data to the Radamsa executable. The strategy of using an external process was chosen to allow the process to be long lived, so Radamsa's internal state can be updated over the course of many inputs. The alternative approach, which other fuzzing projects have taken when adopting Radamsa, is to pull in the `main()` function from the C code (which is generated by the Radamsa Lisp code) and execute `main()` once per input.[27] This is much faster in terms of execution time, because the function is executed within the context of the fuzzer process. This means data does not need to be piped from one process to another and then back again, however it loses a key value of Radamsa, which is that it keeps state and tends to get better as it sees more data.[28] While the implementation in Killerbeez is slower, and this reflects poorly on the metric of executions of the target application per second, it is arguably higher quality mutations in the long run. There was an effort to get Radamsa compiled on Windows as a library, which was painstakingly implemented, only to find that it was about twice as slow as using an external process. The cause of this was not immediately apparent, and the effort was abandoned in favor of developing other features.

Finally there is zzuf, which is yet another application fuzzer, that primarily targets media players, image viewers, and web browsers. As with other mutators which were pulled in from other projects, it has found bugs in production code ranging from audio and video codecs to objdump [57] and nm [57].

Each of the mutators brings diversity to Killerbeez. Different authors are going to frequently have different approaches, and even when the algorithms are similar, there are frequently implementation details which will vary in ways which are sometimes important. Different mutation algorithms will perform differently on various targets. The benefit of being able to switch from one to another easily enables Killerbeez to measure which ones are finding more inputs which trigger new code execution on different targets and at different points in the fuzzing process. A mutator which performs poorly with the initial corpus of inputs may be the best later when a different section of code is unearthed.

4.3 Instrumentation

Killerbeez uses an instrumentation abstraction, to implement the feedback-based portion of the fuzzer. Instrumentation monitors code coverage of the target binary. Feedback-based fuzzing helps expand code coverage by reducing the input set to only those that reach new code. This reduces that the likelihood that multiple inputs will be tested that result in the same code coverage.

The following instrumentation modules have been implemented:

- **Debug** - A naïve Windows-only instrumentation that determines the result of a round of fuzzing via the Windows Debug API.[58]
- **Return Code** - A Linux-only equivalent to the debug instrumentation that uses the `waitpid()` system call to determine the result of a fuzz round.
- **DynamoRIO** - An instrumentation that uses the DynamoRIO project[46] to determine new code paths discovered in a binary.
- **Intel PT** - An instrumentation that uses Hardware-level "Process Tracing"
- **AFL** - An instrumentation injected by a modified version of AFL's compilers (afl-gcc or afl-clang-fast), or via running the executable under a modified version of QEMU[40]

The instrumentation modules monitor, at a minimum, whether a process crashed, exited cleanly, or timed out. More advanced instrumentation modules, such as DynamoRIO, monitors basic block coverage and can inform the fuzzer of new code paths taken in a binary. Instrumentation developers decide what options their module has and whether they will implement optional features. For example, a module can do only lightweight tracking, as is done in AFL, or it can optionally support tracking every basic block executed and each transition. If it can do the slower, more accurate tracing, it is considered to be not only an instrumentation module, but also a tracer. How tracers are used is covered more in section 4.4.

The Debug instrumentation is currently a Windows-only instrumentation which attaches to the target process using the debugging interface and monitors the process for a crash or clean exit. It does not track code coverage, which is commonly referred to as "black box" fuzzing. The driving force behind this module was that there is no reliable way to determine if a process crashed or exited normally on Windows without debugging it. Unlike Unix, the return code does not contain this information, so there is no way to tell the difference between a program that decided to exit with a non-zero status code to indicate an error, and a crash.

The Return Code instrumentation module is similar to the Debug instrumentation in that it does not track code coverage. On Portable Operating System Interface (POSIX) operating systems, the return code of a process is a 32-bit integer. Only the eight least significant bits are provided to the shell, but the full value is available from the `waitpid()` function and macros such as WIFEXITED and WIFSIGNALED can be used to discern between a clean exit with a non-zero exit code and an actual crash.

The DynamoRIO instrumentation module is a modified version of the instrumentation in WinAFL. It requires the user to specify a function which is responsible for loading and processing the input data. At the end of the target function, DynamoRIO will kill the process. Alternatively, this instrumentation supports persistence mode, which allows for multiple inputs to tested without restarting a process. This mode reduces the overhead of restarting the process, and thus increases the number of tests that can be conducted per second. Persistence mode in the DynamoRIO instrumentation is accomplished by resetting the stack and jumping to the beginning of the function again, which may work in test applications, but does not tend to work in real-world software. The typical result is a crash due to global state which is never reset. This includes things like open file handles, allocated memory, application specific state information, etc. The target function must be identified outside of Killerbeez and is typically done manually. By default, an AFL-style bitmap is generated to track code coverage. The module takes options which allow this to be changed to obtain a full trace (see section 4.4 for details on this feature). Other options include a list of libraries which should be covered by instrumentation. This allows things like tracking code coverage in acrord32.dll when fuzzing Adobe Reader [59]. Tracking code coverage in modules is an important feature, because the majority of the input parsing code is encapsulated in a library and recompiling is not an option.

The Intel PT module uses IPT to gather trace information for CPUs which support IPT. This requires a kernel component to manage IPT, but the tracing itself is done in hardware with a modest performance overhead[55, 56]. The current implementation of this instrumentation module [38] only works on Linux, via the "perf" subsystem. Expanding this to support the IPT driver in Windows is planned in the future. Regardless of operating system, the output of IPT relevant for tracing execution in fuzzing are the Taken Not Taken (TNT) and Target IP (TIP) packets. The former tracks "the direction of direct control branches," while the latter records "the target Instruction Pointer (IP) of indirect branches, exceptions, interrupts and other branches or events."[34]

The TNT packets form a bit stream, while the TIP packets contain a series of instruction pointer addresses, which may be compressed if the IP's upper bits match the previous IP value. However, these two packet types are not synchronized. For example, if there are four conditional branches, an indirect jump, and then four more conditional branches, IPT will generate a TIP packet and one byte of TNT packet data with no information about the order in which the TIP and TNT events occurred. To make sense of this data, the executable must be analyzed

to determine the order in which to pull information from the TNT and TIP queues. As this disassembly adds to the performance overhead, the Intel PT instrumentation module does not do it. Instead it takes a hash of the entire TNT bit stream and the entire set of IP addresses in the TIP packets. This does not identify what code was executed, but it does determine if a different code path was taken, as a different code path would result in different packet data, and thus a different hash. Because the packet order is not synchronized between the TNT and TIP streams, hashes are taken of each stream separately and the pair of hashes are used to identify a particular code path.

The IPT instrumentation also supports persistence mode. Persistence mode in the IPT instrumentation is accomplished by modifying the target to repeatedly accept a new input from the fuzzer, call the code to be fuzzed, and reset the target state. While persistence mode in the IPT instrumentation requires source code and manual modifications to the target software, it is much more likely to work properly as compared to the DynamoRIO instrumentation persistence mode.

The only other public fuzzers known to implement Intel PT based tracing are Honggfuzz[42], Richard Johnson's modified version[36] of WinAFL, and kAFL[47]. Honggfuzz does full packet decoding using the Intel's processor trace decoder library[35] which incurs a much higher overhead than the Killerbeez implementation. Richard's fork of WinAFL did full packet decoding at one point, but does not seem to use the trace data at all with the latest commit.[37] Instead, there is a comment which says "FIXME winipt" and the calls to `PtTraceProcessStart()` and `PtTraceProcessStop()` have been commented out, which implies this is still a work in progress. kAFL utilizes a custom packet decoder built specifically to allow efficient parsing of the IPT packets and disassembly of the target executable. As such, kAFL's IPT parser is faster than the Intel processor trace decoder library,[56] but is slower than the approach taken in Killerbeez which refrains from analyzing the target executable.

Finally, we have the AFL instrumentation module, which is based around the instrumentation injected at compile time by afl-gcc or the AFL llvm module. Much of the code was taken directly from AFL and adapted to conform to the Killerbeez instrumentation API. This module has been tested with Linux and macOS, but should work on any POSIX operating system. The injected fork server is a slightly modified version of the implementation in the AFL project. The forkserver is also used by the Intel PT module, which splits the "fork" and "run" actions, as IPT needs to be initialized between these two steps. The original AFL implementation combined these two actions, as they did not have any use case that required them to be separate. The AFL instrumentation modules also implements the persistence mode feature included in AFL. The AFL instrumentation persistence mode is implemented similarly to the IPT instrumentation persistence mode and has similar advantages and disadvantages. AFL's QEMU user mode tracing is also included in Killerbeez, however this mode only works on Linux as QEMU user mode is only available there. QEMU chain caching, which is disabled in AFL, has been enabled in the Killerbeez implementation via the patch made by Andrea Biondo.[39] This patch to QEMU ensures chains are properly tracked and results in a 3-4x improvement in performance.

This puts the Killerbeez implementation of the source-based AFL-style instrumentation equivalent with the original implementation and the QEMU feature significantly faster, showing the advantages of combining the innovations of different authors.

4.4 Tracer

A tracer is an instrumentation module which captures detailed trace information about exactly which basic blocks were executed, along with the transitions between them and implements some optional functions in the Killerbeez instrumentation API which return this information. This coverage information is commonly referred to as nodes and edges.

The DynamoRIO instrumentation module is an example of both a normal instrumentation module and a tracer. By default, it does lightweight tracing to obtain the AFL-style bitmap coverage information and returns an error if it is asked for nodes and edges. The "edges" option can be enabled to switch the module to capture full trace information. Enabling the more accurate tracing mode has a larger overhead, so it is not used for every iteration of fuzzing.

As a counterexample, the Intel PT instrumentation module is currently not a tracer. Without full IPT packet decoding, it is not possible for this module to obtain such detailed information.

Any time trace information is found, the assimilator stores it in the manager's database in a standard format. This is possible because the Killerbeez instrumentation APIs which get nodes and edges require the data to be in a specific format. Any other trace data is allowed to be in any format, as it is passed around as an opaque blob and not consumed by anything other than the instrumentation module which created it. Trace data is accessible via the manager's REST API.

The trace data can later be used to reduce the set of seeds to only include the minimum number of files, or the minimum file size, which hits the maximum amount of code. The concept of minimizing test corpora while maintaining the maximum code coverage dates back to at least October of 2008, when Peach Fuzzer version 2.2 was released, which included the minset tool.[8] There are a number of different algorithms which could be chosen, which is why this is handled by an optional add-in which can be swapped out at will, as shown in Figure 2.

This granular code coverage data can also be used for

weighting seeds based on various algorithms, such as attempting to get to code which is less frequently covered, or targeting a particular piece of code such as a parser which was manually identified or a new piece of code which was identified using automated patch analysis. This would be implemented by the seed selector module from the campaign manager. The most basic seed selection algorithm would weigh all of the seeds equally and go through them in a round-robin fashion.

Determining how often to use the tracer is the responsibility of the job type selector in the campaign manager. This module has the ability, but not the obligation, to take node and edge coverage into account. The decision of when to run the tracer is made based on whether the trace data is necessary for the configured Killerbeez components and how much of a performance impact the tracer will have (as compared to scheduling fuzzing jobs during the same time period). The simplest algorithm would never enable the tracer, which would inhibit all other components from using trace data.

4.5 Picker

Instrumenting all libraries for a real application using a dynamic instrumentation technology is prohibitively slow. Even when using more efficient instrumentation methods, there is a desire to minimize the amount of overhead in instrumentation so more effort can be spent finding bugs instead of performing bookkeeping operations. The Picker automatically determines which libraries should be instrumented so the fuzzer can limit instrumentation to what is interesting, and omit all the other libraries. For deterministic code, this comes with a level of certainty that what is instrumented is, in fact, all of the code which handles the input the fuzzer is sending it. This step is taken when configuring a target, before any fuzzing begins.

The Picker determines the library to instrument by running through all seed values and instrumenting each library separately. If the code is never executed, or the coverage is the same for every input, it implies the library is not important in parsing the input. It is possible that the library handles some aspect of the protocol which was simply never executed by any of the seed inputs, however, with a diverse set of starting inputs, there can be some confidence that nothing is omitted for the list of modules to instrument.

Code which is non-deterministic causes problems with the algorithm above. Tracking down each source of non-determinism and attempting to eliminate it would be a very tedious task. An example of one source of non-determinism was a call to a graphics libraries failing to allocate a surface object. It is difficult to know how to remove this type of non-determinism. Every system call which fails on a regular basis would have to be analyzed, and a decision made on what to do about it. Making it always fail may cut off code paths later which trigger a bug which would be reachable in the program in practice. Repeatedly making the call until it succeeded may also eliminate code paths later, plus makes execution slower at best and an infinite loop at worst.

Instead of trying to force the non-deterministic program to behave in a deterministic fashion, the Picker accepts that the code in question is going to behave erratically and ignores the execution data related to those sections of code. This is done by running the same input through the target N times and finding all of the bytes in the coverage info which vary. By default, N is 10, however it should be large enough that a significant number of executions do not identify any more bytes where the execution varied. The correct value for N will vary from one target to another. The data for Windows Media Player, shown in Figure 4, indicates new non-deterministic code was being found at 325 executions. However, after 10 executions, more than half of the non-deterministic transitions were identified for each of the four libraries.

Choosing the number of executions is a trade-off between spending time up front to get more efficient fuzzing, and getting started more quickly but being less efficient. In addition to determining which libraries to instrument, the identified non-deterministic transitions can be used by some instrumentation modules in determining if an input caused new code paths to be found. This is currently only implemented in the DynamoRIO instrumentation module, but will likely be implemented in the several of the instrumentation modules listed in the Future Work section.

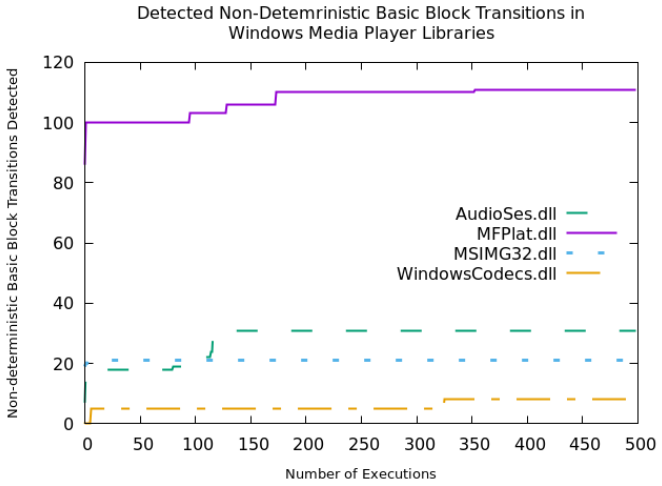

Figure 4: Total Non-deterministic Basic Block Transitions Detected per Execution In Windows Media Player Libraries

The algorithm the Picker uses is not effective with all instrumentation modules. The Picker uses the instrumentation module which will be used in practice, and uses the same options for that module. It then operates on the opaque blob which represents the code coverage information. It does this without any knowledge of the internal

format. Anything which uses the AFL-style bitmap will work fine. This would include the DynamoRIO and AFL instrumentation modules. The IPT instrumentation output is the hashes of the TNT and TIP packets, so any non-determinism will change every byte of the instrumentation data. This will cause the Picker to mask out all bytes of the coverage data, which is not useful. If the Intel PT instrumentation was also a tracer, it would be capable of using an internal format which is compatible with the Picker.

5 Related Work

There are projects which have addressed some of the aspects covered by Killerbeez such as platform independence, distributed fuzzing, leveraging existing tools, and so forth.

Google's OSS-Fuzz[20] addresses scalability by running many fuzzers in parallel, as well as re-using existing code by leveraging tools like Honggfuzz to handle the actual fuzzing. The core fuzzing component of OSS-Fuzz, CluserFuzz, was unpublished and therefore unavailable to anyone outside of Google for the first eight years. On February 7, 2019, ClusterFuzz was released under the Apache Licence (v2).[61] Differences which still remain between OSS-Fuzz and Killerbeez include OSS-Fuzz requiring source code for the target software, and requiring that tests be written to integrate it into the overall system. This adds efficiency, as the test cases can eliminate code like GUI libraries, which is not the real target of the fuzzing, however it is unable to test closed source software. Had Clusterfuzz been open source in 2017, the authors likely would have attempted to extend OSS-Fuzz to also be able to fuzz closed source software and run on computers the user controls instead of using Google's cloud service.

AFL[1] is an open source fuzzer that uses coverage data and genetic algorithms to automatically discover interesting test cases in a target. AFL was designed to be practical; it has a low overhead, is easy to use, and works against real-world software. As such, AFL has become one of the most popular fuzzers and many research projects investigate how to improve AFL. Killerbeez borrows many features from AFL, such as the compiler and QEMU instrumentations. While Killerbeez has also borrowed AFL's mutation strategy, it does not currently include AFL's mutations which mutate based on coverage data. Coverage data is used to avoid wasting time mutating portions of the input that the target does not process. These mutations will be incorporated into Killerbeez in the future. While AFL is a great local fuzzer, it is not easily distributed and cannot easily manage the fuzzing of more complex targets. AFL does not support applications on Windows, and the AFL fork which does support Windows, WinAFL, lacks many of the features of AFL.

Peach Fuzzer[41] now supports distributed fuzzing, modular mutators, and modules for launching apps, is able to do both file and network-based fuzzing, and does work on closed source applications. However, the distributed aspect is only available in the proprietary version of the fuzzer; it does not exist in the community edition, which is open source. There is also no feedback loop for code coverage in either edition. Instead, the input format needs to be manually described in XML files, as does the model for the program state. To alleviate this problem, the company behind Peach Fuzzer is willing to sell access to the definitions they have created.

Honggfuzz[42] is an open source fuzzer which runs on Windows, Linux, macOS, Android, FreeBSD and NetBSD, all using a single code base. It can handle closed source applications, long-running applications such as servers, and will automatically use multiple CPU cores to do fuzzing in parallel. Modularity is achieved by allowing an external program to do the mutation of inputs. While Honggfuzz scales nicely on a single machine, it does not have any built in mechanism to utilize multiple machines. Using Honggfuzz in a larger framework such as OSS-Fuzz takes care of this limitation. In fact, Honggfuzz is a fuzzer which will be likely integrated into Killerbeez in the future, as described in section 6. Honggfuzz has more types of instrumentation than any other fuzzer, including Killerbeez at the time of writing, however none of these work on Windows. The Branch Trace Store (BTS) and IPT instrumentations are only for Linux, as is the hardware-based counters instrumentation which tracks the number of instructions and branches which were executed. There is also compile time instrumentation, but this is only helpful in the case where source code is available, and it can be compiled by GCC or LLVM. It is possible to compile some C/C++ code for windows using LLVM, but anything which requires Microsoft Visual Studio to be compiled will not have any instrumentation. Supporting Windows does not seem to be a priority for Honggfuzz, as it cannot be compiled natively, but only via Cygwin.

Honggfuzz is a great tool, which is why a modified version of it was created which can use all of the Killerbeez modules.[43] Section 6 describes the Linux instrumentation technologies from Honggfuzz planned for integration with Killerbeez. If it is feasible to add the ability to use Killerbeez instrumentation modules, that is another contribution which will be made to the Honggfuzz project.

6 Future Work

The short-term focus will be to incorporate existing technologies from other projects and get them integrated with Killerbeez and running on all the supported platforms. There is ample research, tools, and techniques available now which have yet to be applied in different domains. Once the state of the art has been incorporated, more automation will be the next priority.

Expanding the portability of the instrumentation modules so they work on more operating systems is a high priority. The IPT module currently only works on Linux, but should be able to work on Windows now that Windows IPT driver support has improved. The reverse is true for DynamoRIO, which currently works on Windows but should be able to be ported to run on macOS[5] and Linux without too much difficulty. Pulling in more instrumentation technologies such as Intel's Pin, and Dyninst [31] is also on the list of future work. Wrapping up the future enhancements to instrumentation is making more instrumentations aware of the non-deterministic portions of code which were identified by the Picker. Currently, only the DynamoRIO instrumentation can use this information, but it should be easy to extend this to all of the AFL-bitmap compatible instrumentation modules. This includes the AFL module and the to-be-written Pin and Dyninst modules. The IPT module will not be able to use this data because it does not actually decode the TNT and TIP bitstreams to determine what code is being executed. Adding real-time parsing would slow down the target software significantly, however, this would still be considered if the benefit of handling non-deterministic code appears to be worth the additional overhead which would be incurred. Finally, adding in the Linux-only instrumentation technologies from Honggfuzz [42] is planned, to get the performance boost on software which runs on Linux, especially that which is closed source or not written in either C or C++.

Expanding the selection of drivers to include the ability to monitor dialog box pop-ups on Windows is an area of interest. There are already drivers for the more common input methods, such as files, stdin, and network data, however these should also be expanded to cover kernel functions via syscalls, drivers via IOCTLs, IPC messages, etc. This will make Killerbeez a fuzzer which can handle not only applications on multiple operating systems, but also the kernel of multiple operating systems as well. Currently, this is only theoretically possible with Killerbeez, which is not much help to researchers in the field. Making it supported without any development or modifications will be a big step in helping industry researchers analyze operating system kernels.

The mutation algorithms from several other projects have already been integrated with Killerbeez, however it is desirable to have the ability to write mutators in Python. This will allow pulling in mutators from projects like BrundleFuzz [32] as well as quickly putting together custom target-specific mutators without needing to learn the Killerbeez mutator API, nor even needing a compiler.

In addition to the aforementioned modules on the client side, there are a number of algorithms to pull in from academic publications. This includes the seed energy rating and power schedules from AFLFast [9] and AFLGo.[10] FairFuzz presents another seed selection algorithm to bias seed files toward code segments which are not often executed.[11] The algorithm from PerfFuzz [12] can be incorporated into Killerbeez to find algorithmic complexity vulnerabilities. The research on using estimators and extrapolators to determine if a fuzzing campaign should be stopped, or continue running, can be integrated from Pythia.[13] Instrumentation modules can be updated to implement the collision resistant algorithm from CollAFL.[14] Angora is also on the list of techniques to integrate into Killerbeez, however it is at the end of the queue of improvements due to it needing to be completely re-implemented on account of the authors never releasing the code.[33]

The need to obtain a wide variety of seed values is a weakness of Killerbeez as well as many other fuzzers. The quality of the starting corpus makes a big difference in the efficiency of fuzzing. This issue will be addressed by leveraging existing tools which find new inputs more efficiently, or are able to find inputs which reach code that is unlikely to be hit via mutation.

There are a number of security analysis and fuzzing tools which do not fit into mutators, drivers, nor instrumentation modules. This includes input generation tools, such as Driller [21] and Synfuzz [30]. Integrating Driller can be accomplished by simply scheduling BOINC jobs which run Driller instead of the fuzzer component. The input to the process is still the executable and the code which has been covered thus far, and the output is still inputs which cause new code to be executed, so this will work perfectly with the current system. This allows the BOINC code to be leveraged to handle things like scheduling tasks, dealing with worker nodes which time out, and the campaign manager can deal with deciding how much time to spend drilling versus fuzzing. Preparing a Driller environment which can be easily deployed will require a bit of work, however this should only need to be done one time, not once for each fuzzing target.

Pulling in inputs generated with Synfuzz should be even easier, as the only REST API which should be required is the ability to add files to the corpus. Some automation of setting up Synfuzz may also be possible, but it requires an oracle to determine which inputs are valid or not, which will likely mean it will need to be set up manually, short of new research that allows autonomously detecting such oracles and hooking the appropriate functions.

Finally, expanded support for fuzzers other than the Killerbeez fuzzer is planned. With the client/server architecture which was chosen, there is no reason Killerbeez has to be limited to a single fuzzer. It could easily run Honggfuzz, AFL, WinAFL, or others. Doing so will require additional configuration in BOINC to ensure the previous state is sent to the BOINC clients, as well as a new BOINC assimilator to gather the results.

[5] DynamoRIO's support for macOS is a work in progress.

7 Conclusion

In this paper, we present Killerbeez, a fuzzing framework that integrates many disparate research projects so that they can be used together. This framework allows for independent fuzzing improvements to be reused in a distributed, cross-platform way that they otherwise would not have been able to. Furthermore, Killerbeez allows for each new improvement to be easily evaluated against the existing components, which will expedite the evaluation of new research. The authors have released Killerbeez as open-source software[6] in the hopes that it will allow for the quicker adoption and evaluation of new security research, resulting in a more efficient vulnerability detecting system.

8 Acknowledgments

This research was started through an internal research program funded at GRIMM. The authors would like to thank Brian Schanbacher, Ian Klatzco, and Tommy Chin for their support and contributions. Additionally, the authors would like to thank the many security researchers who have open sourced their prior-research, which eases the integration into Killerbeez.

9 References

[1] Michal Zalewski. *American Fuzzy Lop*. http://lcamtuf.coredump.cx/afl/.

[2] Marc "van Hauser" Heuse. *Collection of Patches to AFL*. https://github.com/vanhauser-thc/afl-patches/.

[3] Ben Nagy. *AFL on OSX*. https://github.com/bnagy/osx-afl-llvm.

[4] Ivan Fratric. *WinAFL - Fork of AFL for Windows*. https://github.com/ivanfratric/winafl

[5] Maksim Shudrak. *winAFL patch to enable network-based apps fuzzing*. https://github.com/mxmssh/netafl

[6] Yan Shoshitaishvili. *Preeny: preload libraries for pwning stuff*. https://github.com/zardus/preeny

[7] University of California. *Berkeley Open Infrastructure for Network Computing*. https://boinc.berkeley.edu/.

[8] Michael Eddington. *Peach Fuzzer version 2.2*. https://sourceforge.net/projects/peachfuzz/files/Peach/2.2/.

[6] https://github.com/grimm-co/killerbeez

[9] Marcel Böhme, Van-Thuan Pham, Abhik Roychoudhury. *Coverage-based Greybox Fuzzing as Markov Chain*. https://www.comp.nus.edu.sg/~mboehme/paper/CCS16.pdf.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, Abhik Roychoudhury. *Directed Greybox Fuzzing*. https://mboehme.github.io/paper/CCS17.pdf.

[11] Caroline Lemieux, Koushik Sen. *FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz Testing Coverage*. https://arxiv.org/pdf/1709.07101.pdf.

[12] Caroline Lemieux, Rohan Padhye, Koushik Sen, Dawn Song. *PerfFuzz: automatically generating pathological inputs*. https://dl.acm.org/citation.cfm?doid=3213846.3213874.

[13] Marcel Böhme. *STADS: Software Testing as Species Discovery*. https://mboehme.github.io/paper/TOSEM18.pdf.

[14] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, Zuoning Chen. *CollAFL: Path Sensitive Fuzzing*. http://chao.100871.net/papers/oakland18.pdf.

[15] Google. *syzkaller - kernel fuzzer*. https://github.com/google/syzkaller.

[16] kernelslacker. *Trinity - Linux system call fuzzer*. https://github.com/kernelslacker/trinity.

[17] MWR Labs. *macOS Kernel Fuzzer*. https://github.com/mwrlabs/OSXFuzz.

[18] eSage Lab. *IOCTL Fuzzer*. https://github.com/Cr4sh/ioctlfuzzer.

[19] Jeremy Brun. *Windows Kernel Drivers fuzzer*. https://github.com/koutto/ioctlbf.

[20] Google. *OSS-Fuzz - Continuous Fuzzing for Open Source Software*. https://github.com/google/oss-fuzz.

[21] Nick Stephens, John Grosen, Christopher Salls, Audrey Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, Giovanni Vigna. *Driller: Augmenting Fuzzing Through Selective Symbolic Execution*. http://www.cs.ucsb.edu/~chris/research/doc/ndss16_driller.pdf.

[22] BOINC. *Increasing Server Capacity*. https://boinc.berkeley.edu/trac/wiki/MultiHost.

[23] Intel Corporation. *Pin - A Dynamic Binary Instrumentation Tool*. https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[24] Aki Helin. *Ni mutator*. https://github.com/aoh/ni.

[25] Aki Helin. *Radamsa - a general-purpose fuzzer*. https://gitlab.com/akihe/radamsa.

[26] Aki Helin. *Radamsa - Some Known Results*. https://gitlab.com/akihe/radamsa/blob/master/README.md#some-known-results.

[27] Trail of Bits. *Grr Radamsa Modifications*. https://github.com/trailofbits/grr/tree/master/third_party/radamsa.

[28] Aki Helin. *Grr Radamsa Modifications Comments*. https://gitlab.com/akihe/radamsa/issues/28#note_77242061.

[29] Sam Hocevar. *zzuf - general purpose fuzzer*. https://github.com/samhocevar/zzuf.

[30] Joe Rozner. *Synfuzz - re-targetable grammar based test case generation*. https://github.com/jrozner/synfuzz.

[31] The University of Wisconsin, University of Maryland. *DyninstAPI: Tools for binary instrumentation, analysis, and modification*. https://dyninst.org/.

[32] Carlos Garcia Prado. *BrundleFuzz - a distributed fuzzer for Windows and Linux using dynamic binary instrumentation*. https://github.com/carlosgprado/BrundleFuzz.

[33] Peng Chen, Hao Chen. *Angora: Efficient Fuzzing by Principled Search*. https://angorafuzzer.github.io/.

[34] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3, p248*. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-volume-3c-system-programming-guide-part-3.

[35] Intel Corporation. *libipt - an Intel(R) Processor Trace decoder library* . https://github.com/01org/processor-trace.

[36] Ivan Fratric, Richard Johnson. *Fork of WinAFL*. https://github.com/intelpt/winafl-intelpt/.

[37] Richard Johnson. *Commit which appears to have removed IPT support*. https://github.com/intelpt/winafl-intelpt/commit/d1e9e560bbaf4e56f6d6bd48672bf691097e86fa.

[38] GRIMM. *Killerbeez IPT Documentation*. https://github.com/grimm-co/killerbeez/blob/master/docs/IPT.md.

[39] Andrea Biondo. *Improving AFL's QEMU mode performance*. https://abiondo.me/2018/09/21/improving-afl-qemu-mode/.

[40] *QEMU*. https://www.qemu.org/.

[41] Peach Tech. *Peach Fuzzer*. https://www.peach.tech/.

[42] Google. *Honggfuzz - A security oriented, feedback-driven, evolutionary, easy-to-use fuzzer with interesting analysis options*. http://honggfuzz.com/.

[43] GRIMM. *Modified version of Honggfuzz which enables it to use Killerbeez mutator modules*. https://github.com/grimm-co/honggfuzz.

[44] Anton Lindqvist. *Fuzzing the OpenBSD kernel*. https://www.openbsd.org/papers/fuzz-slides.pdf.

[45] Oulu University Secure Programming Group. *PROTOS Protocol Genome Project*. https://www.ee.oulu.fi/roles/ouspg/genome.

[46] *DynamoRIO*. http://www.dynamorio.org/.

[47] Schumilo, Sergej and Aschermann, Cornelius and Gawlik, Robert and Schinzel, Sebastian and Holz, Thorsten. *kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels*. https://github.com/RUB-SysSec/kAFL.

[48] Sanjay Rawat et al. *VUzzer: Application-aware Evolutionary Fuzzing*. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/vuzzer-application-aware-evolutionary-fuzzing/.

[49] Joshua Pereyda. *boofuzz: Network Protocol Fuzzing for Humans*. https://github.com/jtpereyda/boofuzz.

[50] GRIMM. *Guided Fuzzing with Driller*. https://blog.grimm-co.com/post/guided-fuzzing-with-driller/.

[51] Audrey Dutcher. *Angr Real World Program Issue*. https://github.com/shellphish/driller/issues/25#issuecomment-288253948.

[52] Mateusz "j00ru" Jurczyk. *Effective File Format Fuzzing*. https://www.blackhat.com/docs/eu-16/materials/eu-16-Jurczyk-Effective-File-Format-Fuzzing-Thoughts-Techniques-And-Results.pdf.

[53] Tavis Ormandy. *Making Software Dumber*. http://taviso.decsystem.org/making_software_dumber.pdf.

[54] Michal Zalewski. *Technical Whitepaper for afl-fuzz*. http://lcamtuf.coredump.cx/afl/technical_details.txt.

[55] James Reinders. *Processor Tracing*. https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing.

[56] Andrea Allievi and Richard Johnson. *Harnessing Intel Processor Trace on Windows for Vulnerability Discovery*. https://conference.hitb.org/hitbsecconf2017ams/materials/D1T1%20-%20Richard%20Johnson%20-%20Harnessing%20Intel%20Processor%20Trace%20on%20Windows%20for%20Vulnerability%20Discovery.pdf.

[57] Free Software Foundation. *GNU Binutils*. https://www.gnu.org/software/binutils/.

[58] Microsoft. *Debugging Functions*. https://docs.microsoft.com/en-us/windows/desktop/debug/debugging-functions.

[59] Adobe. *Adobe Acrobat Reader DC*. https://get.adobe.com/reader/.

[60] Sergey "Shnatsel" Davidoff. *Collection of bugs uncovered by fuzzing Rust code*. https://github.com/rust-fuzz/trophy-case.

[61] Google. *Open sourcing ClusterFuzz*. https://opensource.googleblog.com/2019/02/open-sourcing-clusterfuzz.html.