Auto Off-Target: Enabling Thorough and Scalable Testing for Complex Software Systems

Anonymous Author(s)

ABSTRACT

Software systems powering OS kernels, basebands, bootloaders, firmware, IoT or automotive build the foundation of infrastructure that billions of people rely on every day. Testing these systems is crucial, especially as their complexity grows and they are often written in unsafe languages such as C/C++.

However, testing such complex systems poses significant challenges, e.g., custom hardware for which there is no emulator, or a non-trivial setup of testing and debugging on the target device. As a result, the commonly used testing techniques and tools are not always easily applicable.

An off-target (`OT`) testing is a promising technique which addresses these challenges: part of the code is extracted and adapted to run on a different hardware platform with better tool support, easier debugging and higher test throughput. Unfortunately, since the process of creating an `OT` program has been manual, the technique did not scale well and was mostly used in an ad hoc manner.

In this paper we present a novel complex systems testing approach called *Auto Off-target* (`AoT`). Based on the information extracted from the source code and from the build process, `AoT` can automatically generate `OT` programs in C. `AoT` goes beyond the code generation and provides mechanisms that help to recreate and discover the program state in the `OT` code. The generated `OTs` are self-contained and independent of the original build environment. As a result, pieces of complex or embedded software can be easily run, analyzed, debugged and tested on a standard x86_64 machine.

We evaluate `AoT` on tens of thousands of functions selected from OS kernels, a bootloader and a network stack. We demonstrate that majority of the generated `OTs` can be automatically tested with fuzzing and symbolic execution. We further used `AoT` in a bug finding campaign and discovered seven bugs in the Android `redfin` and `oriole` kernels powering Google Pixel 5 and 6 phones.

KEYWORDS

off-target, testing, fuzzing, symbolic execution

ACM Reference Format:
Anonymous Author(s). 2022. Auto Off-Target: Enabling Thorough and Scalable Testing for Complex Software Systems. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Complex software systems powering OS kernels, firmware, baseband, IoT or automotive are the building blocks and a foundation on which many other systems depend. Ultimately, these systems are used and relied upon by billions of people every day.

The importance and the complexity of these systems call for an increased effort to eradicate software bugs which lead to reliability issues and—even more importantly—open up security holes. Thorough testing is crucial, given that the systems are often written in unsafe languages such as C/C++.

Unfortunately, the use of common automated software testing techniques such as fuzzing [27], [19] or symbolic execution (`symbex`) [11], [12] poses significant challenges in these targets. Usually, there is no executable program or an easy entry point to run. Moreover, the systems often run on a custom hardware, e.g. System-on-Chip (SoC) for a smartphone, while most of the tools are built for x86_64. On-device testing and debugging is often hard due to the target device constraints. For example, we might not be able to use `AddressSanitizer` [32] on a low-memory IoT chip, and it might be hard to debug a bootloader in an infotainment system of a car.

Rehosting [22], [21], [14], [18], [17] addresses these challenges by using virtualization, i.e., running the system or its part under a `VM` which emulates the hardware platform, e.g., in `QEMU` [10]. Although rehosting has the advantage of capturing deep software and hardware interactions, it also has a significant shortcoming: the target hardware needs to be modeled.

While emulation exists for standard CPUs, e.g., `ARM` cores, emulation out of the box is not available for customized and new hardware such as a baseband, SoCs, micro-controllers or DSPs, due to custom architectures or peripherals. For example, no public emulators exist for such popular and important SoCs as Exynos, Snapdragon and M1. Creating and maintaining an emulator is a considerable and non-trivial effort which requires expert knowledge of the emulated hardware platform. Once the system is rehosted we obtain a very good testing environment, but also one that is specific to the emulated target and unlikely to be useful for others.

Off-target (OT) testing is a promising technique in which parts of the source code are extracted and adapted to run on a different hardware platform. As an example lets consider testing a baseband message parser *on-target*: in order to exercise functionality on the target device, we need to setup a network, generate and transmit protocol frames over-the-air. Once a bug is detected, we likely need to halt the test, download logs and start a non-trivial on-device debugging process. On the contrary, an *off-target* test of the same subsystem could be constructed by extracting the parsing code, providing stubs for the remaining functionality, compiling the code on an x86_64 machine and running a fuzzer which would automatically generate and feed the test data into the parser.

The off-target approach has a considerable advantage of focusing testing efforts on a security-critical functionality, e.g., complex

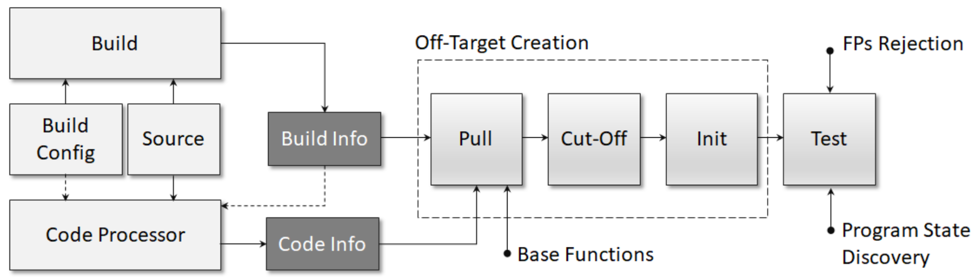

Figure 1: An overview of the AoT approach.

input format parsers, in an environment which is much better equipped for debugging and root cause analysis and has a higher test throughput. In fact, we often do not need to run the entire system in order to discover critical security issues.

Unfortunately, the process of creating an OT program is manual and challenging. First, the code in scope needs to be extracted. Next, the code dependencies, e.g., types, need to be resolved by pulling in more code. Finally, the off-target approach comes with an inherent challenge: since we extract a part of a system and run it elsewhere, the original program state is missing. This involves memory allocations and the exact values in memory which need to be provided by the user. Missing allocations or incorrect constraints on the program state result in *false positives* (FPs) - bugs which are only present in the off-target code but not in the target system. As a result of all the mentioned challenges, the technique has mostly been used in an ad hoc manner for creating one-off OTs.

In this paper we present a novel complex system testing approach called *Auto Off-Target*, or AoT for short. AoT can automatically generate off-target programs in C based on information extracted from the source code and the build process. This is our main contribution, however, AoT goes beyond the code generation and also helps to tackle the missing program state challenge. AoT generates memory allocation code, leverages dynamic testing techniques to discover program state and uses data flow analysis to reject false positives.

The generated OT code is independent of the original environment and decoupled from the build process, i.e., no special knowledge of the build flags is required to compile and run the OT. As a result, pieces of complex or embedded software can be easily run, analyzed, debugged and tested on a standard x86_64 machine.

We evaluate AoT on tens of thousands of functions selected from OS kernels, a bootloader and a network stack. AoT was able to run fuzzing and symbolic execution—two state-of-the-art software testing techniques—in majority of the generated OTs out of the box. We further used AoT in a bug finding campaign and discovered seven bugs in the kernels powering Google Pixel 5 and 6 phones.

The rest of the paper is organized as follows: in §2 we present an overview of the AoT technique, in §3 we discuss the architecture and the implementation of our prototype, in §4 we evaluate the prototype and we discuss limitations in §5, in §6 we discuss related work and we conclude in §7.

2 OVERVIEW

Before we describe AoT in detail, we define the terminology and provide an overview of the technique.

As mentioned in the introduction, by off-target creation we mean the process of extracting a part of system's source code to run it on another target. We call the original execution environment the *old host* and the destination execution environment the *new host* as it will host the extracted part[1]. The system we extract code from is called the *target* and the extracted part itself is called the *off-target (OT)*. For example, in the mentioned baseband message parser testing scenario we could have the following associations: the *old host* is an Android smartphone, the *target* is the baseband protocol stack, the *off-target* is a selected message parsing functionality and the *new host* is an x86_64 Linux machine on which we test the OT.

A high-level overview of the proposed AoT testing approach is presented in Fig. 1. Two fundamental building blocks required to run AoT are *build information* (*Build Info*) and *code information* (*Code Info*).

The build information includes a list of compiled files, linked modules and compiler flags used, as well as the information about dependencies between the modules, source and intermediate files. This information is extracted during a full build of the target system. It is worth noting that the build needs to be performed *only once per target* as AoT operates entirely on the previously collected data.

The code information includes the source code of functions, types, global variables (globals), references among types, globals and functions as well as code metadata including variable assignments, casts and structural type member dereferences. The code information is generated from the original source files by the *Code Processor* (see §3) with the help of build information or build configuration (*Build Config*) alone if the meta build system used provides enough details. The information required by the code processor is a list of compiled files and the build flags being used.

An OT is generated by taking a list of *Base Functions* that represent the functionality we would like to test, and pulling recursively (*Pull*) all the functions that are called by them, along with the required types and globals. Since AoT operates on precise information, only the necessary code is pulled in, unlike when including entire headers which could contain code the OT does not use.

Next, a decision is made about which functions should be kept and which should be left out based on the *cut-off algorithm* used (*Cut-Off*). The functions included entirely in the OT are called *internal* and those left out are called *external* as illustrated in Fig. 2. For the external functions AoT generates *function stubs*, i.e., instances

[1]There might also be cases in which we wish to extract part of the code and run it in the same execution environment.

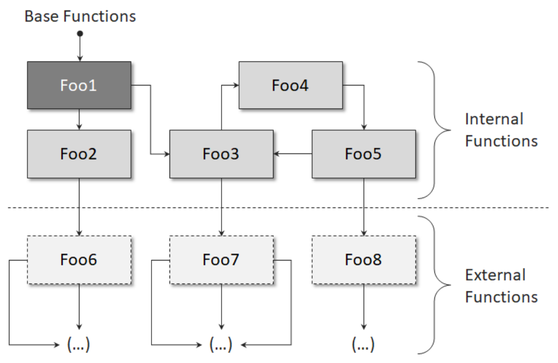

Figure 2: OT creation: base, internal and external functions.

of the functions without a body. Since the original bodies of the external functions are removed, the functions further called by them are not included in the OT, unless they are internal.

As mentioned in §1, the off-target testing approach comes with an inherent challenge: since a part of the code is extracted, the generated OT is missing the original system state, e.g., the values of global variables or function parameters that are normally set on the running target. This is an important problem as the lack of proper memory initialization could prevent correct execution (and testing) of the OT and result in *false positives* (FPs). As an example let's consider the code of function foo presented in Listing 1.

Failing to allocate memory for the pointer y results in a crash on dereferencing y->member on line 12. If the pointer is correctly initialized on the target, this crash is an FP.

```
01 #define SIZE 10
02 #define SIZE_A 15
03 globtype_t g[SIZE];
04 rettype_t arr[SIZE_A];
05
06 rettype_t foo (size_t x, struct B* y) {
07   if (x != SIZE) return 0;
08   for (int i = 1; i < x; ++i) {
09     int selector = bar(i) + g[i];
10     if (selector < SIZE_A) return arr[selector];
11   }
12   return y->member;
13 }
```

Listing 1: Code example for program state initialization.

AoT implements three automated approaches to help tackle the open challenge of recreating program state in the OT code: smart init, state discovery and FPs rejection.

Smart init (*Init* in Fig. 1), which we discuss in detail in §3, is a heuristic based on static analysis. Smart init discovers which types, globals and structure fields are likely used in the OT and initializes them. The initialization is performed both for non-pointer and pointer variables for which the memory is dynamically allocated.

AoT makes it possible to further refine function parameters initialization via a user-provided JSON *data init file*. The user can select the function and the parameters of interest by name and specify, e.g.: the value of an integer parameter, the size of an array, whether the array is null-terminated or the order of parameters initialization. We envision that such a file can be created along the code review and updated as more information about the particular target is discovered.

While correct allocation of program state is a prerequisite to executing the code at all, it is also important to initialize the state with the right values in order to get a meaningful execution. On the target system the initialization happens before the given function is called, maybe even during the device bootup, but in the OT it needs to be performed explicitly.

In the example from Listing 1, the parameter x has to be equal to SIZE in order for the execution to go past line 7. Also the value of selector, which is based on the values retrieved from function bar and global g, has to be smaller than SIZE_A in order for the execution to return at line 10.

In order to enable the execution of OT in the presence of unknown program state, AoT performs *program state discovery* by using automated dynamic testing techniques: symbolic execution and fuzzing. The key idea is that we treat as a test input not only the user data, but also the program state, excluding pointers, which are explicitly initialized. By using the mentioned techniques AoT lets the program explore as many execution paths as possible—some of them correct, some not. Since the program state is fuzzed, it is encoded in the test cases generated in the process and can be used for initialization in the OT. Even if the program state initialization is not perfect and results in FPs, the OT should still be able to execute some correct behaviors of the tested code. By leveraging automated software testing techniques AoT is able to execute and explore the OT code without having *any* prior knowledge of the original values of the program state.

In our approach we perform a conservative *over-approximation* of data, i.e., we potentially consider more values than feasible on target. We allow the values to be only constrained by the code found in the OT, however, in the target system additional constraints might be imposed. For example, if a parameter p is never validated in function f, we assume it can take any value. However, if on the target the function f is always called *after* the value of p is validated, by limiting our analysis in the OT to only the function f we consider some values of p that would not be allowed on the target. Those values might in turn lead to FPs.

When analyzing FPs it is important to distinguish between the values related to system state, e.g., structures describing file nodes in the kernel, and the ones controlled by the user, e.g., a buffer passed from the user space to the kernel. This is especially important in security testing: an overflow caused by a variable controlled by the user could be exploited and therefore is a security issue.

In AoT, the user can explicitly *tag* a variable by applying a *taint* [30] to it. The tagging process can also be automated via the data init file described earlier, e.g., we can specify that for a class of functions from an API, a certain buffer parameter always comes from the user.

AoT implements two automated mechanisms to tackle the FPs. The first is based on data flow analysis and rejects the bug-revealing test inputs for which no tagged data is detected near the location of a bug. The second in an on-device check mechanism for Android: we take the crashing test inputs, extract the user payload and verify it on the device with an automatically synthesized test program.

Let's go back to our example from Listing 1 and assume that the user can only control the value of y->member, which is tagged. In line 10 we dereference the array arr with the index selector. Since the index is a signed integer, it could be negative, which would result in an underflow. However at that point in the execution, the tagged data remains untouched. Since the bug is not caused by user-controlled data, it is ignored.

Once an OT is generated, it can be debugged, analyzed or tested (*Test* in Fig. 1). AoT is independent of the testing technique used and it implements support for fuzzing and symbex.

The generated OT might need to be adjusted by the user. The adjustment usually relates to implementing stubs and refining the data initialization. AoT comes with scripts that guide the user through the process—they run symbex and fuzzing on OT, collect code coverage and report on the stubs that were encountered during the execution. Based on that information the user should be able to quickly identify areas for improvement. Due to all required types, globals and functions being present in the OT, this manual task of tuning the code should be moderately easy. In many cases the generated OT is good enough to start fuzzing straight away without any changes as we report in §4.

AoT automates the laborious process of creating OT which usually involves multiple iterations of pulling in code, checking if it compiles, pulling in more code, initializing state, etc. As a result, with AoT the user can focus on creating meaningful tests rather than on creating the OT itself. Moreover, we can select any part of a very complex system and thoroughly test in a way similar to unit testing.

3 IMPLEMENTATION

In this section we discuss the implementation details of the AoT prototype. Fig. 3 shows a high-level overview of the AoT architecture. AoT design is based on *Code Aware Services* (CAS) [9] infrastructure for which we implemented the AoT application.

3.1 CAS Infrastructure

Code Aware Services (CAS) infrastructure is a framework designed to provide build and code information for other tools. CAS consists of two components: *Build Awareness Service* and *Code Database*.

Build Awareness Service (BAS) collects and provides information on all interconnections and dependencies between source code components at the scale of an entire product code. It can operate on very large source trees, e.g., ≈1.1 million source files (82GB in size) for Pixel 6 AOSP version, including Android platform and the Linux kernel, a scale which is usual for a full mobile product. BAS provides selected information to AoT through exported JSON files.

BAS, which is presented in Fig. 4, consists of four modules: *Build Tracer*, *Build Database*, *Build System Plugins*, and BAS *Core*.

Build Tracer observes the build process and tracks the executed programs as well as the referenced files. It is based on a specialized

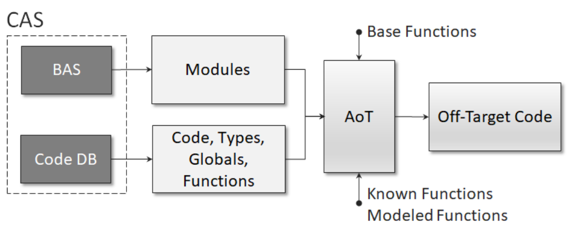

Figure 3: AoT architecture.

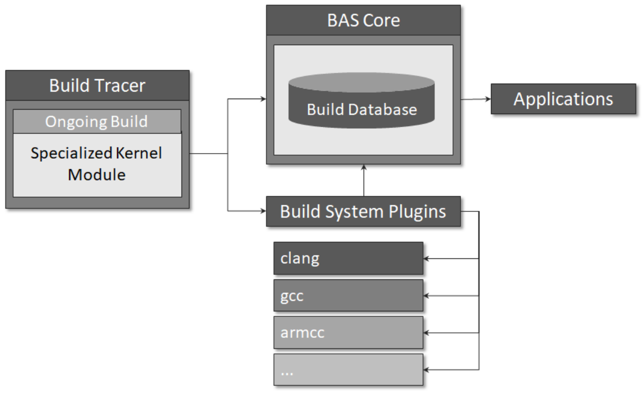

Figure 4: Build Awareness Service architecture.

kernel module which guarantees very low overhead. Build Tracer does not require any specific configuration or modification of the build system. The user only needs to run the build command under the control of the tracer, i.e.:

```
$ etrace build_command <parameters>
```

Build Tracer is fully transparent to the underlying build system, therefore, all build systems on Linux are supported out of the box.

As a result of running target's build with the tracer, a *Build Database* is created. It contains information about executed processes, precisely the execve system calls (syscalls), and all opened files. This data is further processed to transform the raw syscall information into a format better suited for information retrieval.

An important step in Build Database post-processing is parsing compilation commands by using compiler-specific *Build System Plugins*, one per each compiler. Currently supported compilers are gcc, clang and armcc, which cover a broad range of potential build targets, including complex Android mobile products.

BAS Core is a service that exposes information from the Build Database to external clients—the *Applications*. It defines an API which can be used to issue queries to the Build Database.

Some of the features of BAS used by AoT are:

- Getting all linked modules created during the build along with dependencies between modules.
- Getting reverse module dependency information for a given source file (used for Code DB generation and the cut-off algorithm).
- Getting a list of source files used to build a specified linked module.
- Getting compilation commands for a set of compiled files (used during the creation of Code DB).

BAS is mainly written in C, and Python. The size of C code is around 20KLOC and the size of Python code is around 2.5KLOC. Occasionally some fragments are written in C++ with the size estimated at around 3KLOC. There is also around 1KLOC of SQL code involved in database processing.

In the last few years there has been a major advancement in compiler technologies due to the creation of the extensible libclang library. This led to the development of a number of tools that operate directly on the source code by using the compiler infrastructure, such as static analyzers [24], sanitizers [20], [32], custom passes for code transformation, source code indexing tools [36], etc.

The `libclang` library makes it possible to use selected functionalities of the full-fledged C/C++ compiler in an external application. As opposed to writing a working parser from scratch, by using `libclang` it is possible to transform source code into an AST (Abstract Syntax Tree) form which is much easier to process.

Code Database (Code DB), which is the second major component of `CAS`, is generated by using the *Code Processor* based on `libclang`. The *Code Processor* uses the `clang` parser to extract various pieces of information from the original source and stores the extracted data in a `JSON`-like database for further processing. In order to run, the processor needs to know all commands used to compile the original source files. This information can be retrieved from `BAS` or from the original meta build system, e.g., `CMake`, if available.

Code DB contains information on all defined types, including dependencies between the types. Based on this information `AoT` can emit source code that fully defines a given type without using any external references such as header files. Furthermore, Code DB contains a list of all defined globals. For each global it stores: the type of the global, the initialization code and the references to other globals and functions. Code DB also contains a list of all defined functions. For each function it stores the function body and a number of attributes, e.g., called functions and function pointers, referenced types, defined variables, taint information for function arguments, variables used in `switch` and `if` statements, and more.

Importantly, Code DB stores the initialization and assignment of function pointers. This information is helpful when working with source code that heavily uses indirect function calls as a mode of operation, e.g., the Linux kernel driver framework.

The process of creating the Code DB for a single linked module is presented in Fig. 5[2]. First, all the source files that are compiled for a specific module are listed by using the Build Database of `BAS`. Second, compilation commands for the files are generated from the Build Database[3]. Third, the Code Processor uses the commands to parse each source file and generates Code DB for each translation unit. This step can be performed in parallel on a multi-core machine. Information extraction happens at the level of internal `clang` parser's structures which makes the process relatively easy. Finally, all intermediate instances are merged into a single final Code DB instance that describes source attributes of the entire linked module, e.g., Linux kernel linked image, `vmlinux`.

For complex systems such as Android (e.g., ≈60k executed compilations for full Android AOSP `oriole` + kernel) the amount of source code data which needs to be stored and processed can significantly affect performance. In Code DB only selected information is extracted from the AST, which greatly decreases its size. We find the existing contents of `BAS` and Code DB sufficient for multiple applications, however, if more information is needed, Clang Processor can be extended to collect it.

Clang Processor is mainly written in `C++` with additional `Python` support code. The size of `C++` code is around 23KLOC and the size of `Python` code is around 3KLOC. We use code printers from `clang` and an open source `JSON` parser.

[2]Please note that Code DB can also be created for multiple modules.
[3]All tools that use `libclang` framework use a special `JSON` database that contains detailed compilation switches used to parse each source file.

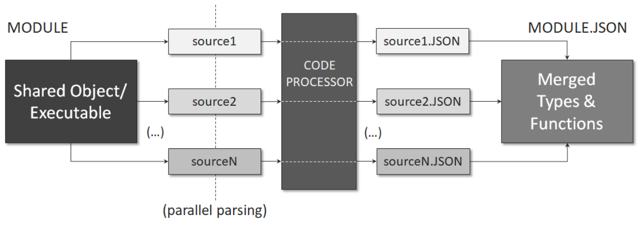

Figure 5: Code Database creation process.

3.2 AoT Core

`AoT` makes extensive use of the information provided by `CAS` components `BAS` and Code DB. The information from `CAS` is provided to `AoT` via `JSON` files. `AoT` then stores the information in a `MongoDB` [28] instance. The database import needs to happen *only once* per target, not per `AoT` invocation.

It is important to note that since `AoT` is based on `CAS`, it takes into account exactly the code that is compiled into the final product configuration. Moreover, `AoT` is completely separated from the build process – it depends solely on the database information so it can run without having access to the original source tree, which could be very large for complex systems. Thanks to storing data in `MongoDB`, multiple distributed `AoT` instances can be executed simultaneously.

We implemented the core functionality of `AoT` as a Python program of about 8KLOC. The current prototype supports C code only but we plan to extend it to support C++. As an input, `AoT` takes a list of base functions for which we would like to generate the `OT`, as well as a list of *known functions* which we would like to avoid pulling in, e.g., common functions from the C standard library such as `memcpy`. The assumption is that the `OT` will use the versions of those functions provided by the standard C library on the new host. The user can also define *modeled functions* by providing own implementations which will be used in the `OT`, e.g. own version of `kmalloc`. For each base function, `AoT` recursively reconstructs all the dependencies such as called functions, globals and precise type information by querying the Code DB. The code pulled in by `AoT` is *preprocessed*, i.e., all macros, `#define` and `#ifdef` statements are resolved.

`AoT` implements several cut-off algorithms to distinguish between internal and external functions: based on the same compiled module, source files, source directories and a list of functions. The `CAS` data is queried as a part of the cut-off process to establish which functions belong to the same modules, directories or files. Functions that are listed as *known* or *modeled* are cut off too.

If the user wishes to include functions that have been marked as external, it can be done with a granularity of a function, file or source directory level. Since assembly is not portable across hardware architectures, we treat all functions containing assembly as external by default[4]. One notable source of inline assembly are macros. Because macros are included directly in the function body by the preprocessor and `OT`s are generated from a preprocessed code, it would be challenging to filter the macros in a textual form. To tackle that issue we extended `CAS` with an option to replace macros with user-defined snippets.

[4]The default behavior can be overridden with a runtime option.

Once all code dependencies are reconstructed, AoT generates the OT program code. The generated programs are pure C applications that: (1) are free of #ifdef statements dependent on the build configuration, (2) contain all the necessary types, (3) provide stubs for all the external functions and (4) provide smart initialization of the program state.

The OT is generated in a directory containing multiple source files and a makefile. By default, AoT generates the following binaries for an OT: asan, dfsan, ubsan, afl, KLEE, coverage, debug and a standard native application. The functions and globals are spread across the generated source files in a way that corresponds to their original locations. For simplicity, only a single header file is generated. AoT generates #ifdef guards in case a name clash of types occurs in the header.

It is worth noting that compilation commands for an OT are very simple—they do not need any include paths or specific compiler options as the generated code is preprocessed and independent of the original old host environment. As a result, the OT code is also much easier to parse and process with cross-references in an IDE; it also builds fast. A developer can change the code, rebuild the OT and see the results much faster as compared to the full build and uploading the binary to the target system.

An OT is generated with a main() function from which the base functions are called and with the program state initialization code for globals and function parameters.

AoT comes with a library which abstracts memory management and fuzzing details away from the user. As an example let's consider two functions:

-
  ```
  int aot_memory_init_ptr(void** ptr,
      unsigned long size, unsigned long count,
      int fuzz, const char* name)
  ```

-
  ```
  int fuzz_that_data(void* ptr, void* src,
      unsigned long size, const char* name)
  ```

The first function is used to allocate memory for pointers. It takes the pointer address, the size of the pointer type, the number of elements to allocate (> 1 for arrays). The *fuzz* parameter selects if the memory is to be filled with values coming from a fuzzer (fuzz==1) and an optional parameter *name* allows to assign a name to the allocated memory.

The second function is used to explicitly fill size of bytes under memory pointed by ptr with data generated by a fuzzer (src denotes the original source of data).

Note that which fuzzer is selected and how the data is passed from the fuzzer to the buffer is completely transparent to the user. AoT currently implements support for afl/afl++ [19] fuzzer and for KLEE [11] symbex engine. The OT makefile makes it possible to compile binaries suited for each supported testing approach.

AoT implements a hybrid testing approach [33] for OTs: first it runs symbex which is well suited to discover program state. Then, the generated test cases are used as initial inputs to a fuzzer. Note that both testing techniques can also be used in isolation.

Helper scripts provided with AoT make it possible to automatically reject FPs coming from an incomplete data initialization, collect testing code coverage, analyze issues with asan [32], pull in more functions in case a stub for a function originally marked as external causes FPs and verify the crashes on a device (Android only).

3.3 Code Synthesis

AoT synthesizes source code files based on the type/function/global data retrieved from Code DB. Rather than copy-pasting existing source files, AoT re-creates them from scratch by pulling in only the code that is needed.

In the C programming language there is a dependency among types, globals and functions. Types are used in functions and globals, but types can also reference other types, functions and globals. Globals may reference functions and other globals. In order for a program to compile, AoT needs to discover and pull in recursively all types, functions and globals. This is possible thanks to the information about cross-references provided by the Code DB.

Type definitions might be nested, e.g., a structure may be defined inside another structure. Typedefs can contain type definitions too. When such nested types are generated, AoT makes sure that only the containing type is generated, as its body will automatically contain the other type definitions. AoT generates forward declarations and takes care of the type definition order by applying topological sort on the emitted types.

AoT generates special wrappers for the static functions which need to be called from another file, e.g., from the test driver file that contains the main() function.

In stubs, AoT generates a log message, commented original code and a return statement. For builtin types stubs return zero, for pointers stubs return a value from a reserved memory region that will be easy to recognize, and for structure types returned by value, stubs return an object of a given type initialized to zero.

3.4 Smart Init

The *smart init* mechanism is used to allocate memory for the program state, as mentioned in §2. Before discussing the details, let's consider a motivating example presented in Listing 2.

```
01 int foo (void* x) {
02   struct A* a = (struct A*)x;
03   struct B* b = (struct B*)a->member1;
04   struct C* c = b - offsetof(struct C*, member2);
```

Listing 2: Smart init challenges example.

The function takes a single pointer argument x of type void*. In order to initialize memory, AoT needs to know how much space to allocate and which members of structural types are used in the OT[5]. Unfortunately, this cannot be always done by only looking at parameter's type as we might allocate too little memory. In line 2, x is cast to type struct A*. Then, in line 3, we reference a pointer member of struct A and cast it to struct B*. Finally in line 4, the cast member is used in an offsetof operator to retrieve a pointer to struct C. To sum up, in order to correctly initialize memory for this example, we need to allocate x as struct A. Moreover, we need to allocate an object y of type struct C and set x->member1 to point to y->member2.

The smart init mechanism is based on a lightweight static analysis of the following events: type casts (explicit and implicit), member accesses for structural types and the use of offsetof operator. We choose these three events since they can help discover the real type of a variable passed to a function. The events are stored in the

[5]Initializing all the members would be ideal, however, it is unrealistic in practice as some structures are very complex, e.g., see the task_struct type in the Linux kernel.

Code DB for each function in the AST parsing order of the compiler (Code Processor). This order roughly reflects the sequential order of the events in the code if the code is read line by line[6]. For example, the event describing member access in line 3 will happen before the event corresponding to the `offsetof` operator use in line 4.

The insight behind the smart init heuristic is that for each input parameter of a function, AoT parses the ordered trace of events in a way similar to reading the code line by line. Starting from the initial parameter type, AoT modifies the type accordingly when encountering the mentioned event types.

It is a common pattern in the structural types in C/C++ that an array and its size are represented by two separate members of a type. In order to correctly initialize program state it is necessary to capture that relationship and to make sure the integer members used to dereference arrays are correctly constrained. AoT performs a static analysis which matches structure members by name, checks the use of integer members in comparisons (ifs, loops) and constant-size arrays, as well as the use of constant offsets in arrays.

3.5 Rejecting FPs with dfsan

AoT uses dfsan to reject FPs caused by an incomplete state initialization. Each failing test case is run through a dfsan-instrumented binary which logs `load`, `store` and `cmp` events for tagged data only (the user input). Each event is logged with a call trace. The call traces are then matched against the original issue call trace with a degree of fuzziness. If a match is not found, the test case is rejected as an FP.

4 EVALUATION

In order to evaluate AoT we selected four target systems:

- **T1**: Android `oriole` Linux kernel, which is arguably one of the most complex and thoroughly tested pieces of publicly available software, powering Google Pixel 6 phone [8],
- **T2**: The Little Kernel Embedded Operating System (`lk`) [5], which is a non-Linux operating system based on microkernel architecture,
- **T3**: Das U-Boot (`uboot`) [15], which is a bootloader for embedded devices,
- **T4**: The IUH module [7] of the Osmocom project [6], which implements the IUH interface for femtocell communication from a 3GPP standard [23].

All the selected systems represent complex software which is in large parts written in C. We selected the projects spanning operating systems, embedded software, bootloaders and telecommunications to illustrate wide applicability of AoT. The largest target by far is T1, as it contains over 166k functions; T2 contains 1,732 functions, T3—4,771 functions and T4—3,507 functions.

In this section we first provide the details of our test setup. Next, we report the performance statistics for the generation of CAS and OTs and provide the OT testing results. Finally, we describe a manual assessment of OTs for a selected set of user space-facing functions in Android kernel that led to the discovery of seven bugs.

[6]Whenever that is not the case, AoT reorders items in the metadata to reflect a natural flow of the code.

4.1 Test Setup

For the evaluation we used two racks of `x86_64` machines running Linux Ubuntu 20.04 LTS. The first rack (R1) comprises eight nodes, each with 64 CPU threads and 128GB of RAM, and the second rack (R2) comprises eight nodes, each with 128 CPU threads and 256GB of RAM. The nodes use AMD Ryzen Threadripper 2990WX/3990X CPUs and are connected through a 10Gbps network[7]. For the targets build and the creation of CAS databases we used a single node. When creating CAS we limit the number of threads to 32, therefore, the differences across R1 and R2 nodes should be negligible.

For the AoT evaluation we use GNU `parallel` [34] with the following distributed setup: we pick a single main node from R1 to run `parallel` and store the results. In addition, we use twelve worker nodes: five from R1 and seven from R2. We assign half of the available execution threads, i.e., 32 on the nodes from R1 and 64 on the nodes from R2—a total of 608 parallel execution threads—and let `parallel` spread the jobs across nodes. Each thread is assigned the following tasks: (1) generate OT, (2) compile, run sanity check and testing, (3) send the results to the main node, (4) cleanup.

As the main OT compiler we use `clang 10`, for KLEE we use `clang 9` and for compiling the dfsan binary we use `clang 15` which we extended to support `sscanf`. Our version of KLEE is based on commit `b73c45` from KLEE version `2.3-pre`. We use `afl++` version `4.00c`, Python version `3.8`, MongoDB version `4.4`, and for measuring lines of code we use `cloc` [2] version `1.82`.

We use the following versions of the target systems: branch `android-gs-raviole-5.10-android12-qpr1-d` for T1, commit `77fa08` for T2, commit `537159` for T3 and commit `1f6c11` for T4.

4.2 Performance Stats

Before we can run AoT, we first need to perform the target system build, prepare CAS and import the data to a MongoDB instance, as discussed in §3. Table 1 presents the times required for each step.

Table 1: Performance stats (in seconds). T_{Build} is the time required to build a target, $T_{Trace\Delta}$ is the additional time needed when the CAS build tracer is turned on, $T_{\mathtt{BAS}}$ and T_{CodeDB} are times required to create BAS and Code DB, respectively. $T_{\mathtt{MongoDB}}$ is the time required to import CAS data to MongoDB.

Target	T_{Build}	$T_{Trace\Delta}$	$T_{\mathtt{BAS}}$	T_{CodeDB}	$T_{\mathtt{MongoDB}}$
T1	161.65	10.42	146.38	1609.69	372.78
T2	1.19	5.57	2.27	6.22	19.83
T3	2.65	4.88	7.51	21.87	24.14
T4	12.84	8.09	43.65	11.01	22.47

As we can see, the overhead imposed by the Build Tracer is very small: the tracer adds at most 10.42s to the build time and the measured values include a few seconds of tracer startup overhead. Also the times required to prepare CAS databases and to import data to MongoDB are relatively low. The total overhead required to prepare data for AoT in the largest target T1 is about 36 minutes. We believe it is reasonable as the steps need to be performed only once per target build.

[7]We run MongoDB on four Intel i7-7820X machines with 128GB of RAM.

The data size in MongoDB for T1–T4 is: 3.76GB, 21.7MB, 64MB and 41.3MB, respectively.

In order to illustrate the scalability of CAS, let's consider a few exemplary statistics collected for a recent Android AOSP `oriole`.

For the entire platform and kernel, BAS is able to detect 2,476 shared libraries (.so files) and 347 executables that were produced during the build. For `vmlinux`, which is the linked kernel image, there are 20,168 file dependencies in total, among which there are 2,668 compiled files and 3,188 header files. The reverse module dependency feature makes it possible to retrieve the list of linked modules that depend on a specific file. For example, header file `include/hardware/bluetooth.h` is used in 30 modules from the entire platform. By using BAS we can obtain a compilation commands file for a selected module, e.g., there are 3,711 compilation commands required to generate `vmlinux` and all loadable kernel modules.

Code DB for the entire `oriole` kernel source tree contains information about 168,309 functions, including 132,316 static and 69,736 inline functions. There are 86,299 distinct types[8], including 30,216 structures, 5,096 enumerations and 6,014 constant arrays. Moreover, there are 91,466 globals, out of which 68,516 have internal linkage. All globals use 5,761 types. Code DB is extracted from 3,711 source files and 220 modules (`vmlinux` and 219 kernel modules). There are 20,930 initializers for function pointer members of global structures, which allow to track function calls dispatched through a pointer.

4.3 Generating and Testing OTs

In this section we provide the results of a fully automatic test performed on tens of thousands of OTs. The purpose of the test is to demonstrate: (1) the scalability of AoT, (2) the ability to generate compiling OT code, (3) the effectiveness of the smart init and program state discovery in creating immediately usable OTs.

We select a random sample of functions in each target: 50k for T1, 1k for T2, 2k for T3 and T4. We then run AoT on each function from the sample and treat the function as a single entry point to the OT. We use the default cut-off algorithm based on the same linked module. Since assembly is not portable across architectures we skip functions containing assembly[9]. Next, we build each OT and run KLEE and `afl++` on the ones which compile successfully, allowing up to 1h testing per tool. AoT generates dictionaries for `afl++` with `int`, `float`, `char` and `string` literals extracted from Code DB.

Out of the four test targets we spent the most time working with T1. As a result, we developed a custom implementation of stubs for a few most commonly used functions such as `kmalloc`, `kfree` or `mutex_lock`. We further provided a data init file automating the program state initialization for most common classes of user space-facing functions: *read, write, show, store* and *ioctl*. No additional stubs or data init file development was done for T2–T4 and we were able to run those benchmarks after a few minor adjustments.

Table 2 presents OT creation statistics for the automatic test. Across all targets, AoT created over 52k OTs, excluding the cases in which we encountered assembly as presented in the fourth column. An average time needed to generate an OT ranges from below a second to about 1.5 minute. An average OT was a small program ranging from 1.3KLOC to 7.4KLOC, depending on a target. AoT usually created below 10 source files. It is worth noting the number of types required on average can be as high as 1,736 in T1. Similarly, while T2–T4 did not use many structure types, T1 on average used over 347 of them. This illustrates how difficult it might be to manually satisfy OT type dependencies. The generated OTs included on average a few globals and external functions (stubs) and from a few to 20 functions from the target.

We were able to compile most of the created OTs: from 85% in T4 up to 94% in T1 and T3. The high successful compilation rate and the absolute number of the OTs that compile, which is over 49k, show that AoT can generate compiling OT code at scale and on highly complex targets. To the best of our knowledge it is the first tool which can achieve that task.

There were cases in which AoT failed to generate OTs: 1.8% in T1 and 2.9% in T3. Also, in a number of cases the generated OT did not compile. The usual reason for compilation failure is pulling in too few or too many types, wrong ordering or uncommon language features/code constructs. Given the prototype stage of AoT we are confident the numbers can be improved.

Smaller sizes of the generated OTs as compared to the original target make it easy to focus on the selected part of functionality: less code means that it is easier to process and index in an IDE. Importantly, it also makes code coverage results more meaningful. Traditionally, code coverage is measured w.r.t. an entire target's binary. On the contrary, when testing with an OT code, the user is in control of how much code is added. Moreover, the code is added based on call dependencies, so we only see functions called by some other functions in the OT. As a result, code coverage should represent meaningful results for the functionality we are interested to test, not for the entire target.

Another important feature associated with smaller code footprint is the compilation speed. As presented in Table 2, for T1 it takes 21s on average to build an OT. The build includes several targets: native and debug binaries, `asan`, `dfsan`, `ubsan`, `afl`, KLEE and `coverage`. The user can quickly modify, rebuild and debug the code. On the contrary, the same process without OT comprises rebuilding the target image, flashing the image on the device, booting and debugging the code on target, possibly with further reboots. According to our measurement, flashing the image and booting the device alone takes 182s on a Pixel 5 phone, which is an order of magnitude more than the average rebuild time of OT.

Table 3 presents OT testing stats. The primary purpose of the exercise is not to find bugs but to demonstrate that through automated state discovery we can achieve a usable program state initialization. For all targets in over 90% of cases the smart init mechanism worked well enough to test the OT and collect code coverage. As a sanity test case we use an all-zeros buffer, i.e., whenever fuzzer data is requested, bytes filled with zeros are returned. The test is performed before we run KLEE and `afl++`.

In order to illustrate how much of the code pulled from the target was covered, the coverage was measured excluding the main test driver file, AoT libraries and files with stub definitions. Even though the test was fully automatic, AoT achieved OT code coverage from 46.4% on T1, up to 92.2% in T4. Moreover, both KLEE and `afl++` managed to find a few test cases on average. This illustrates that the

[8]Code DB treats types with qualifiers as distinct, e.g., `Type_t != const Type_t`.
[9]Although T4 is compiled for x86_64 which is our destination architecture, we exclude assembly there too for consistency with T1—T3.

Table 2: OT Creation stats. *Sample Size* is the number of functions in the random sample. *No asm* is the number of functions from the sample that do not contain assembly. *Created* OTs is the number of successfully generated OTs. T_{Create} is an average OT creation time. *LOC* presents the average size of OT: just for the code that we pull and including the AoT functions. *Files, Types, Struct Types, Globals* and *Funcs* show the code stats (averages) for OTs. *Builds* is the number of OTs that compile (in brackets as a percentage of generated OTs) and T_{Build} is an average time required to build an OT.

Target	*Sample Size*	*No asm*	*Created* OTs	T_{Create}	*LOC*		*Files*	*Types*	*Struct Types*	*Globals*	*Funcs*		*Builds*	T_{Build}
					Pulled	*Total*					*Int*	*Ext*		
T1	50k	48,835	47,970	85.7s	6,041	7,425	15	1,736	347	13	20	14	45,132 (94%)	21s
T2	1k	806	806	0.6s	235	1268	4	35	4	1	6	2	692 (86%)	3s
T3	2k	1,927	1,871	2.3s	410	1469	4	61	10	2	8	2	1,777 (94%)	4s
T4	2k	2,000	2,000	0.9s	284	1605	6	96	8	10	3	14	1,713 (85%)	3s

quality of the generated OTs was high enough to enable effective code testing and that OT code can be explored without or with only little (see *data init file*) knowledge about the system state.

4.4 Using AoT in a Bug Finding Campaign

In order to evaluate AoT in a human-in-the-loop scenario we report results of code reviews aided by AoT performed on Android redfin and oriole kernels powering Google Pixel 5 and 6 phones, respectively. We focused on functions which have a direct interaction with the user space from five common interfaces used in device drivers: *read, write, show, store* and *ioctl*. In addition, for the redfin kernel we selected all remaining functions that send/receive data to/from the user space and are not directly called by any other functions, as such functions are likely entry points to the kernel. In total, our list comprises 4,584 functions for redfin and 4,812 for oriole.

We leveraged the knowledge of the interface we test and used a data init file that automated program state initialization for the five function classes. In addition, we made use of the *tagging* feature to reject FPs.

For each function we generated an OT and ran tests with KLEE and afl++ (afl for redfin), allowing up to 1h per tool on oriole and 2h on redfin. The tests of redfin were performed on an earlier version of AoT which used afl and rejected FPs with a modified KLEE.

Overall, the FPs rejection mechanism filtered out majority of issues reported by the testing tools. AoT rejected 2,679 FPs across 588 OTs in redfin and 3,486 FPs across 1,227 OTs in oriole.

Table 3: OT Testing Stats. *Testable* OTs is the number of OTs for which tests were run and coverage data collected (in brackets we provide the value as the percentage of OTs that compile). The remaining columns present average numbers per OT.

Target	*Testable* OTs	*Coverage*		*Test cases*	
		Funcs	*Lines*	KLEE	afl++
T1	43,622 (96%)	53.1%	46.4%	2.4	4.5
T2	659 (95%)	75.9%	66.9%	3.8	7.2
T3	1,665 (93%)	73.3%	66.4%	3.5	7.4
T4	1,710 (99%)	98.9%	92.2%	4.5	3.4

As a result, we were left with 131 of out 4,584 OTs with potential bugs to investigate in redfin and 181 out of 4,812 OTs in oriole. The OTs for redfin were investigated by a team of security engineers who found five bugs in a few days: we discovered CVE-2021-39652, rediscovered CVE-2020-13143 and found three out-of-bounds reads. We reported all bugs to Google, except for CVE-2020-13143, which we confirmed is fixed in a newer release. The OTs for oriole were investigated by a single engineer in less than a week. Two issues were reported and are currently under disclosure. This experiment demonstrates that the tagging mechanism provides an effective way to filter out FPs and reduce the number of cases for manual triaging to a manageable size. Our experience shows that AoT is a useful bug finding aid.

When working on a promising OT that initially produces FPs, the user needs to adjust the code, often by improving the program state initialization. Our experience shows that the manual adjustment part usually takes between 5m and 2h. In rare cases, when a deeper understanding is required, it can take more time. However, once the OT is adjusted it can be thoroughly tested, to the level hardly achievable on target within a reasonable time budget. For example, in some of OTs we tested, afl executed the code millions of times.

AoT enables deep testing of complex software. Even when no bug is discovered, the ability to test thoroughly improves the confidence in the code. Notably, AoT enables symbex on complex software systems which we believe is a step towards selectively verifying functions for being free of certain classes of bugs, such as buffer overflows. This property is enabled by symbex but only if it can extensively explore the entire program search space, which is infeasible if the program is too large.

5 LIMITATIONS

As every build-based tool, CAS requires access to the source code and the user needs to be able to build the target system. As mentioned in §3, we currently support clang, gcc and armcc compilers, however, adding support for additional compilers is possible and is moderately easy. AoT currently supports C only; the support for C++ is being implemented in the CAS infrastructure. CAS supports all language features supported by clang, but compiler-specific language extensions implemented by other compilers are not supported.

We believe that CAS should cover the majority of the system and firmware targets in C, but we make a note that non-Linux build systems are not supported.

By default, code that contains assembly is not included, since AoT cannot automatically translate assembly across architectures. If needed, such code can be imported by using a runtime flag.

A limitation of OT testing approach as compared to on-target tests is that we do not test the entire system, so certain interactions across modules or across hardware and software might be missed. Similarly, we might discover FPs, i.e., bugs which would not be possible in the target system due to constraints which are absent in the OT. It is up to the quality of program state initialization and the stubs to correctly model the original environment. We believe that the developers who work on the code are best suited to quickly distinguish between true bugs and FPs and provide the right implementation for the stubs.

Correct and precise automatic program state initialization in OTs is a challenging open problem. We believe that improving on the smart init mechanism could significantly reduce the FPs rate in OTs.

6 RELATED WORK

The off-target approach has been used informally as an ad hoc tool for enabling fuzzing for complex systems code. To the best of our knowledge AoT is the first tool to automate this process. Research on extracting parts of a system has been done in the context of client-side code of web applications [25] based on collected execution traces for particular use cases. In AoT, which operates on systems code rather than web applications, an arbitrary part of a complex system can be extracted and compiled on a different host.

In rehosting [17], [22], [21], [14], [18] the code is run off-target via emulation of the old host system. This approach is well suited for low-level code that depends on hardware. In AoT we target pieces of code which are not dependent on hardware and generate programs that can be compiled on x86_64.

Recent years have seen development of compiler-aided tools that can parse C/C++ source code and store the collected information in a database.

The clang compiler is able to dump the program's AST in a tree-like text format for further processing. The disadvantage of operating on that level is a large amount of text to be parsed: even though the parsing is much easier than with the original code, it is still significantly complex as the parser needs to support a very large number of AST node types. Other clang tools like clang-query [1] allow to look for specific nodes in the AST.

semmle [31] and joern [37], [38] are full-fledged engines that save parsed source code data in a database and use dedicated query languages to look for specific information in the AST. They both provide domain-specific languages to issue queries about the code. There is an additional learning curve associated with getting familiar with the languages. CAS-based applications such as AoT operate directly on a human-readable JSON format which is easy to process in most popular programming languages. The query logic is shifted towards the applications in the CAS framework.

There are several solutions for gathering build information with compile commands. CMake [3] and ninja [26] build systems make it possible to generate compilation database along the build. The bear tool [29] can intercept all program executions during the build, but only for dynamically linked executables and the file access is not tracked. In order to find dependencies between files it is necessary to use other tools based on strace [16] or inotify [4]. However, these tools incur much overhead: e.g., our previous build tracer based on strace had an overhead of ≈54% for full AOSP build when building on 8 cores and an overhead exceeding 1000% on 48 cores. Our current Build Tracer is implemented as a specialized kernel module which makes use of the Linux tracing infrastructure. It can trace commands and file accesses with very low overhead: for the full Android AOSP oriole + kernel build the overhead is ≈6% when using 64 cores.

Fuzzing [27], [19], [35] and symbolic execution [11], [13], [12] are very successful and widely adopted dynamic testing techniques, especially in the field of security. Tools such as afl [40], syzkaller [35] or KLEE [11] have found many bugs in popular libraries and operating systems. In order to merge the advantages of fuzzing and symbex, hybrid approaches such as Driller [33] have been proposed. AoT is not a new fuzzer, instead it enables fuzzing, symbex, unit-like testing and debugging of complex systems code, an activity that would normally require special on-target setup or virtualization.

CSmith [39] synthesizes correct programs in C with the aim of testing compilers. On the other hand, AoT extracts selected parts of the original target code to analyze, test and debug it off-target.

7 CONCLUSIONS AND FUTURE WORK

One of the open challenges in testing complex software systems is to provide strong software quality guarantees at scale and easily apply popular testing techniques. In this paper we presented AoT, a novel approach for an automatic creation of off-target programs.

AoT makes it possible to select arbitrary parts of the original target code, extract them and compile on a different target as pure C programs without dependencies and at scale. AoT goes beyond this main contribution and implements several techniques to address the challenge of missing program state in the OT programs. The smart init and automated state discovery mechanisms help to allocate and initialize program state, while the FPs rejection mechanism based on data flow analysis helps to filter out issues not caused by the user input data.

We evaluated AoT on tens of thousands of functions from various complex and embedded targets and demonstrated that on average 86% generated OTs can be tested out of the box with popular tools. By using AoT in a bug finding campaign, we discovered seven bugs in two kernels powering Google Pixel phones.

We are not aware of any existing system that can generate compiling and testable OT code similarly to AoT. AoT enables and encourages unit-like testing, debugging and use of sophisticated techniques such as symbex on complex systems code. We believe it is a step towards deeper and thorough software testing.

In the future work we plan to focus on the program state initialization, support for C++ and leveraging the code information for structure-aware fuzzing. We plan to extend the data init file to include global variables, types and deeper parameter dependencies. We also plan to extend AoT with an ability to initialize program state by using memory dumps performed on target. Although this approach would be device-specific, we hope to be able to automatically retrieve and use precise program state data in the OTs.

Finally, we released core engines of CAS and AoT to open source.

REFERENCES

[1] 2021. clang-query. https://devblogs.microsoft.com/cppblog/exploring-clang-tooling-part-2-examining-the-clang-ast-with-clang-query/

[2] 2021. cloc. https://github.com/AlDanial/cloc

[3] 2021. CMake. https://cmake.org/

[4] 2021. inotify – monitoring filesystem events. https://man7.org/linux/man-pages/man7/inotify.7.html

[5] 2021. The Little Kernel Embedded Operating System. https://github.com/littlekernel/lk

[6] 2021. Osmocom. http://osmocom.org/

[7] 2021. Osmocom Iuh and HNB-GW implementation. https://github.com/osmocom/osmo-iuh

[8] 2022. Android Oriole branch used. https://android.googlesource.com/kernel/gs/+/refs/heads/android-gs-raviole-5.10-android12-qpr1-d

[9] Anonymized. 2018. Build Awareness Service. In *Unpublished*.

[10] Fabrice Bellard. 2021. QEMU website. https://www.qemu.org/

[11] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[12] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[13] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices* 46, 3 (2011), 265–278.

[14] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware re-hosting through abstraction layer emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1201–1218.

[15] Wolfgang Denk. 2021. Das U-Boot—the Universal Boot Loader. https://www.denx.de/wiki/U-Boot/

[16] Levin Dmitry. 2021. strace – trace system calls and signals. https://strace.io/

[17] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. 2021. SoK: Enabling Security Analyses of Embedded Systems via Rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 687–701.

[18] Bo Feng, Alejandro Mera, and Long Lu. 2020. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 1237–1254.

[19] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association.

[20] Google. 2021. Google Sanitizers. https://github.com/google/sanitizers

[21] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. 2019. Toward the analysis of embedded firmware through automated re-hosting. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2019)*. 135–150.

[22] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. 2020. {PARTEMU}: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*. 789–806.

[23] Douglas N Knisely, Takahito Yoshizawa, and Frank Favichia. 2009. Standardization of femtocells in 3GPP. *IEEE Communications magazine* 47, 9 (2009), 68–75.

[24] LLVM. 2021. Clang Static Analyzer. https://clang-analyzer.llvm.org/

[25] Josip Maras, Jan Carlson, and Ivica Crnkovi. 2012. Extracting client-side web application code. In *Proceedings of the 21st international conference on World Wide Web*. 819–828.

[26] Evan Martin. 2020. Ninja. https://ninja-build.org/

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

[28] Inc. MongoDB. 2021. MongoDB. https://www.mongodb.com/

[29] László Nagy. 2021. Build EAR (BEAR). https://github.com/rizsotto/Bear

[30] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[31] Semmle. 2021. Semmle - Code Analysis Platform for Securing Software. https://semmle.com/

[32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. Addresssanitizer: A fast address sanity checker. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 309–318.

[33] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.

[34] Ole Tange et al. 2011. Gnu parallel-the command-line power tool. *The USENIX Magazine* 36, 1 (2011), 42–47.

[35] Dmitry Vyukov. 2021. Syzkaller. https://github.com/google/syzkaller

[36] WoboqGmbH. 2021. Code Browser by Woboq for C and C++. https://woboq.com/codebrowser.html

[37] Fabian Yamaguchi. 2021. Joern – Open-Source Code Querying Engine. https://joern.io/

[38] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604.

[39] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[40] Michal Zalewski. 2021. AFL website. https://lcamtuf.coredump.cx/afl/