Automating Patching of Vulnerable Open-Source Software Versions in Application Binaries

Ruian Duan†, Ashish Bijlani†, Yang Ji†, Omar Alrawi†, Yiyuan Xiong*, Moses Ike†,
Brendan Saltaformaggio,† and Wenke Lee†
{ruian, ashish.bijlani, yang.ji, alrawi, mosesjike}@gatech.edu, yiyxio@gmail.com
brendan@ece.gatech.edu, wenke@cc.gatech.edu
† *Georgia Institute of Technology*, * *Peking University*

***Abstract*—Mobile application developers rely heavily on open-source software (OSS) to offload common functionalities such as the implementation of protocols and media format playback. Over the past years, several vulnerabilities have been found in popular open-source libraries like `OpenSSL` and `FFmpeg`. Mobile applications that include such libraries inherit these flaws, which make them vulnerable. Fortunately, the open-source community is responsive and patches are made available within days. However, mobile application developers are often left unaware of these flaws. The App Security Improvement Program (ASIP) is a commendable effort by Google to notify application developers of these flaws, but recent work has shown that many developers do not act on this information.**

Our work addresses vulnerable mobile applications through automatic binary patching from *source patches* provided by the OSS maintainers and without involving the developers. We propose novel techniques to overcome difficult challenges like patching feasibility analysis, source-code-to-binary-code matching, and in-memory patching. Our technique uses a novel variability-aware approach, which we implement as OSSPatcher. We evaluated OSSPatcher with 39 OSS and a collection of 1,000 Android applications using their vulnerable versions. OSSPatcher generated 675 function-level patches that fixed the affected mobile applications without breaking their binary code. Further, we evaluated 10 vulnerabilities in popular apps such as Chrome with public exploits, which OSSPatcher was able to mitigate and thwart their exploitation.

I. Introduction

It is a common practice for software developers to use well-adapted third-party libraries to accelerate the application development process. These third-party libraries, like any traditional software, contain implementation bugs that are found by security researchers. Large open-source libraries have active developers who support, maintain, and occasionally fix software bugs. Unfortunately, mobile application developers who rely on these libraries must remain vigilant of bug disclosures affecting their application.

Mobile application developers must track third-party libraries, maintain awareness of disclosed bugs, apply patches while ensuring backward compatibility, and test for unintended side-effects. For the Android platform, Google has initiated the App Security Improvement Program (ASIP) [21] to notify developers of vulnerable third-party libraries in use. Unfortunately, many developers, as OSSPolice [15] and LibScout [4] show, do not update or patch their application, which leaves end-users exposed. Android developers mainly use Java and C/C++ [1] libraries. While Derr et al. [14] show that vulnerable Java libraries can be fixed by library-level update, their C/C++ counterparts, which contain many more documented security bugs in the National Vulnerability Database (NVD), are still not addressed. There are ample efforts to secure mobile platforms and applications through automated patching, but they are limited by the type of bugs and availability of compiled patches. For example, PatchDroid [42] relies on the availability of a *compiled patch*, which is applied in-memory dynamically. Similarly, techniques for platforms like Docker [46] and Android OS [7] also rely on *compiled patches*. Other approaches [48, 76] are limited to a specific type of bugs, such as buffer overflow. Some approaches [3, 8, 73, 79] assume that debugging symbols and build configuration options for compiled applications are readily available, where in reality they are not.

A more effective approach would automatically patch from source code, where patches to OSS are readily available. There are several challenges to patching from source code, such as identifying build configuration for the target applications, matching source code to binary code for missing debug symbols, and addressing statically linked libraries. In addition to these challenges, automatic patching might introduce unintended side-effects that hinder the target mobile application. Based on a recent OSS study by Li et al. [34], the security patches that are applied to a vulnerable code base are localized and are limited in their side-effect, unlike non-security patches. This insight implies that automatic mobile application patching for security-related bugs may be an attainable effort.

To this end, we propose a novel technique to automatically patch vulnerable mobile applications from the source code provided by the effected OSS libraries. Our approach is a layered pipeline that builds function-level binary patches from source code and performs in-memory patching on vulnerable mobile applications. To address source code patch generation challenges, we perform a feasibility analysis to identify function-level patches, then build a variability-aware abstract syntax tree (VAST) to enable further analysis. Using the VAST, we map function addresses and identify build configurations for the

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23126
www.ndss-symposium.org

```
@@ static int ssl_scan_serverhello_tlsext(SSL *s, ...
#ifndef OPENSSL_NO_EC
...
      *al = TLS1_AD_DECODE_ERROR;
      return 0;
    }
-   s->session->tlsext_ecpointformatlist_length = 0;
-   if (s->session->tlsext_ecpointformatlist != NULL)
-     OPENSSL_free(s->session->tlsext_ecpointformatlist);
+   if (!s->hit)
      {
...
+       s->session->tlsext_ecpointformatlist_length
+         = ecpointformatlist_length;
+       memcpy(s->session->tlsext_ecpointformatlist,
+         sdata, ecpointformatlist_length);
      }
-   s->session->tlsext_ecpointformatlist_length
-     = ecpointformatlist_length;
-   memcpy(s->session->tlsext_ecpointformatlist,
-     sdata, ecpointformatlist_length);
...
#endif
```

Fig. 1: Source patch for `CVE-2014-3509` of `OpenSSL`.

target library in the mobile application. We then compile only the patched vulnerable functions from the source code using the derived build configurations. Additionally, we overcome in-memory patching challenges with statically linked libraries using a rerouting approach to ensure the mobile application remains functional. We implement these innovative techniques in a system we call OSSPATCHER.

To evaluate our source-code-to-binary-code matching algorithms, we prepare a labeled dataset and show that OSSPATCHER can identify function addresses and build configuration with 82% recall and 95% precision. We apply OSSPATCHER on 39 OSS and identify 675 feasible patches. We use these patches to fix 1,000 affected Android applications. OSSPATCHER performs in-memory patching and incurs negligible memory and performance overhead, demonstrating the practicality of our system. Further, we test OSSPATCHER capabilities on 10 vulnerabilities with public exploits and show that exploitation of the affected mobile applications is no longer possible.

II. CHALLENGES

OSSPATCHER faces several challenges when automating patching of vulnerable OSS versions in application binaries without developers' involvement. We present them along with a real-world patch for `CVE-2014-3509` of `OpenSSL` shown in Figure 1.

A. Configurable OSS Variants

For the purpose of portability in different deployment platforms and configurations, software product line engineering provides efficient means to implement variable software. By selecting from a set of features, a developer can generate different software variants from a common product-line implementation. C/C++ OSS employs this technique to allow developers to configure OSS for their own use. We refer to such configurations as *variability*. Variability in C/C++ OSS is achieved using conditional directives (e.g., `#ifdef`, `#ifndef`, `#if`) to selectively compile certain parts of source code, or through a build system (e.g., `kconfig` [56] for building the Linux kernel) to selectively compile certain source files. The number of variants could be *exponential* to the number of features. For instance, the recent `OpenSSL` stable release (version 1.1.0h), contains more than 160 preprocessor-based configuration options for enabling/disabling various ciphers, algorithms and protocol extensions, from which countless variants could exist.

However, this variability causes challenges for automatic binary patching because the patching needs to identify the variant and follow the same variant as before in building the patch otherwise it can break the functionality of the application. Although we have access to the source of the to-be-patched function, the patch target is closed-source binary software, so before OSSPATCHER builds the patched function from the open-source code, we need to first figure out the configuration options that were used in the original building of the software and enforce the same in building the patch. For example, the vulnerable code in Figure 1 is enabled only if the macro `OPENSSL_NO_EC` is not defined, which requires OSSPATCHER to infer value of `OPENSSL_NO_EC`. Moreover, the function `ssl_scan_serverhello_tlsext` contains 5 conditional macros (i.e., 32 function variants), which if ignored may lead to vulnerability identification failures and disruption to the patches.

To reverse-engineer OSS feature configs previously used by app developers, one can either compile all variants of the OSS and perform binary-to-binary analysis, or perform variability-aware source-to-binary analysis. Since the former does not scale due to the exponential number of OSS variants with regard to features, OSSPATCHER adopts the latter solution, i.e., builds VAST for OSS and performs source-to-binary analysis (§III-C).

B. Statically Linked Binaries

The build dependencies between the app and OSS sources blur their boundaries, which increases the difficulty of patching the desired library. For example, several C/C++ native libraries can be statically linked to a single library first, then finally linked to the application. Due to the blurred boundary of libraries in such cases, it is hard to pinpoint the original vulnerable library if we would perform library-level patching. In addition, proprietary code can also be statically linked into these libraries, which further adds to the complexity of reverse engineering library boundaries. In such multi-binary files, features across multiple library components are effectively fused into a superset and boundaries among them are hard to be identified.

In the case of statically linked binaries, individual vulnerable libraries cannot be upgraded without replacing all their embracing libraries, which requires more fine-grained patching schemes. Based on the observation that security fixes are localized and small [3, 34], OSSPATCHER performs *function-level* patching, instead of library-level. Our key idea is to identify the function boundary, rather than library boundary, and replace vulnerable functions with patched ones.

C. Stripped Binaries

Stripped builds raise significant challenges in designing a patching system for application binaries. Currently, both

major kernel [3, 8, 73] and userspace [42] patching solutions use symbols to locate vulnerable functions. However, a recent study [15] shows that 98.9% of native libraries in Android apps are stripped and only exported symbols (non-static) remain to allow other programs to link to them dynamically. Other symbols, such as static functions and variables, are not visible and thus require extra efforts to locate them. Moreover, even non-static symbols can be hidden when app developers statically link multiple libraries together. This happens when the option `-fvisibility=hidden` is used during compilation. To work with stripped binaries, OSSPATCHER performs a series of matching analyses to identify the location of the vulnerable function in the application binary (§III-B), so it can perform in-memory patching against it. In fact, we can choose either in-memory patching or binary rewriting for our purpose. We apply in-memory patching in our current implementation because it allows safe reversion of the patch on exception and helps in debugging.

III. DESIGN

A. Goals and Assumptions

We envision OSSPATCHER as an automated system that fixes n-day OSS vulnerabilities in app binaries using publicly available source patches. As mentioned in §II, OSSPATCHER must consider OSS variants and perform function-level matching with no access to debugging symbols in app binaries. While prototyping OSSPATCHER, we focused on fixing uses of vulnerable OSS written in C/C++ for Android apps, but the design is generic and also applies to other Linux-based apps and programming languages, such as Java. OSSPATCHER consists of two modules that are deployed separately: *server* that automatically adapts and compiles source patches for app binaries containing vulnerable OSS versions, and *client* that downloads and applies binary patches to installed applications.

OSSPATCHER assumes that sources of apps are not publicly available, and that developers compile OSS directly from their release versions without tampering with OSS source code. OSSPATCHER also assumes that information from NVD, such as the specified vulnerable versions and the corresponding patching commits are accurate[1]. To this end, we set the following goals:

- OSSPATCHER can accurately identify vulnerable functions and its patch-related config options for patching.
- OSSPATCHER can automatically generate binary patches and perform non-disruptive patch injection.

The workflow of OSSPATCHER is depicted in Figure 2. To meet the aforementioned goals, we have designed three major components in OSSPATCHER: `Analyzer`, `Matcher` and `Patcher`. Analyzer analyzes source patches for their feasibility and converts vulnerable functions that can be patched into VAST. Matcher performs variability-aware source-to-binary comparison to identify function addresses, config options, and variable addresses. Patcher generates patched libraries from source patches and performs in-memory patch injection. The rest of this section elaborates these components.

[1] Patch analysis tools such as UCKLEE [50] or regression tests can be used to further validate correctness of patches. We consider testing of publicly known patches orthogonal to OSSPATCHER.

B. Feasibility Analysis

In this work, we focus only on automatically applying patches where source code changes are contained entirely within functions. We believe this choice does not affect the effectiveness of OSSPATCHER as many security patches are small and localized according to a recent study [34], and thus can be handled by OSSPATCHER. Furthermore, this is similar in scope to previous major patching systems, including Ksplice [3], Karma [8] and PatchDroid [42]. Therefore, the first step in our feasibility analysis is to determine whether the OSS patch can be successfully applied by OSSPATCHER. This process filters out the non-localized patches with large range of code changes (e.g., change to a `struct` definition). As reported in §V, OSSPATCHER can handle over 60% of all OSS patches we crawled from public OSS repos.

A naive approach to check if modifications are solely inside a function would be to use regular expressions to identify functions in source files and compare their source ranges against code changes in patches. But this can be error-prone because of comments and preprocessor directives [41]. Thus, we designed a systematic feasibility analyzer to perform multi-pass source range analysis. Given an OSS patch commit, we parse the affected files using the default config. Since the source code is conditionally compiled, some parts may be skipped due to compile-time options (e.g., `define`), we therefore collect semantic information as well as skipped source ranges. If code changes in patches do not overlap with skipped source ranges, we then check if they are inside functions to report feasibility. If code changes are inside skipped source ranges, we use our SMT-based expression analyzer to find a config combination that enables the skipped ranges and re-parse source files. Finally, we apply the qualified patches to old versions and ensure that they are compatible by performing several checks, such as patch context matching and function signature verification. We break feasibility analysis into three relatively independent tasks, namely, *source range analysis*, *expression analysis* and *version analysis*, and describe them in the following.

Source Range Analysis. The source range analysis finds the semantic context for code changes in a patch, based on which we determine whether the patch is feasible or not. Specifically, we consider the following change types and their combinations as feasible: 1) add, remove, or modify functions, 2) add, remove, or modify comments and empty lines, 3) add or remove extern entries, macro definitions, structs, and inclusion directives. However, this list is preliminary, and other types can be incrementally added as needed. For example, `LibPNG` patch `188eb6b` for `CVE-2010-1205` adds several *typedef* entries to update versions in addition to function-level changes. Since *typedef* statements do not change program semantics and can be ignored, `188eb6b` should be considered as a feasible patch, though currently classified as infeasible.

To perform source range analysis, OSSPATCHER first clones the OSS and checks out a patch commit. Since the exponential amount of OSS variants inhibit brute-force approaches §II-A, OSSPATCHER starts from any one of the many OSS variants, collects skipped source ranges, and builds the corresponding AST. OSSPATCHER then checks semantic context for code changes in patches to decide feasibility. If changes are inside skipped source ranges, OSSPATCHER performs expression

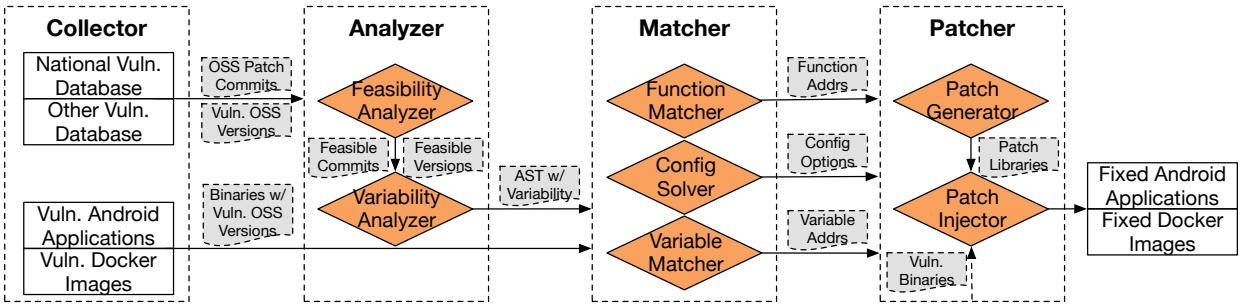


Fig. 2: OSSPATCHER architecture and workflow.

```
defined(PNG_FLOATING_POINT_SUPPORTED) &&
!defined(PNG_FIXED_POINT_MACRO_SUPPORTED) &&
(defined(PNG_gAMA_SUPPORTED) ||
  defined(PNG_cHRM_SUPPORTED) ||
  defined(PNG_sCAL_SUPPORTED) ||
  defined(PNG_READ_BACKGROUND_SUPPORTED) ||
  defined(PNG_READ_RGB_TO_GRAY_SUPPORTED)) ||
(defined(PNG_sCAL_SUPPORTED) &&
  defined(PNG_FLOATING_ARITHMETIC_SUPPORTED))
```

Fig. 3: An expression used in `LibPNG`.

```
#undef PNG_FIXED_POINT_MACRO_SUPPORTED
#undef PNG_FLOATING_ARITHMETIC_SUPPORTED
#undef PNG_READ_BACKGROUND_SUPPORTED
#undef PNG_cHRM_SUPPORTED
#undef PNG_gAMA_SUPPORTED
#undef PNG_sCAL_SUPPORTED
#define PNG_FLOATING_POINT_SUPPORTED
#define PNG_READ_RGB_TO_GRAY_SUPPORTED
```

Fig. 4: A solution to the expression in Figure 3.

analysis to enable such ranges and invokes source range analysis again to decide their feasibility.

Expression Analysis. Conditional preprocessor directives are used in OSS to enable/disable certain parts of source code. We refer to conditions in these directives as *expressions*. If code changes in patches overlap with skipped source ranges, we need to find out a configuration to enable skipped parts for further source range analysis. Nevertheless, expressions in conditional directives such as `#if` and `#elif` can be very complex. For example, Figure 3 shows an expression in `LibPNG` which uses 9 macros. According to the C Preprocessor standard [19], *expression* is a C expression of integer type, and may contain integer constants, character constants, arithmetic operators, identifiers and macro calls. Intuitively, existing compiler frameworks such as *LLVM* [31] and *GCC* [61] should be able to analyze expressions. However, we found that they are designed to speed up the build process; the expressions evaluated as false are simply skipped and not analyzed further. For example, if `PNG_FLOATING_POINT_SUPPORTED` in Figure 3 is not defined, compilers consider the expression as false and skip the rest. Consequently, a LLVM plugin will not emit details needed by OSSPATCHER.

We, therefore, design an analyzer to analyze expressions and solve them using a satisfiability modulo theories (SMT) solver. The analyzer performs lexing and parsing to generate AST from expressions, which is similar to simple calculators, except for support of macro calls and undefined variables. It then converts AST into intermediate code, resolves macro calls using collected macro definitions and symbolizes variables using CVC4 SMT solver [5]. During symbolization, *defined* is a reserved function that checks if a variable (macro) is defined or not, and imposes an implicit constraint that a variable can not have a value unless defined. To interpret this constraint, we create a boolean symbol to represent whether a variable is defined or not and add a constraint that if a variable is not defined, then its value is invalid (NaN). For example, expression `defined(FOO) && FOO > 5` is interpreted as:

$$FOO_{\text{defined}} \wedge FOO > 5 \wedge \neg FOO_{\text{defined}} \implies FOO = NaN$$

In addition, since conditional directives can be nested, our expression analyzer also supports constraint concatenation. We run the analyzer on Figure 3 and present one solution in Figure 4. The solution is represented as `#define` and `#undef` directives and can be used to enable skipped source ranges. OSSPATCHER then invokes *source range analysis* to parse source files and check whether code changes are feasible.

Version Analysis. OSSPATCHER also needs to check whether the source patch changes are compatible with old vulnerable OSS versions. Patches generated by git [67] use the unified diff format [68], which provides metadata, such as changed lines, files, and context lines around these changes. While applying patches, context lines are used to identify locations of changes. If context does not match, patches are rejected. For example, the 4-6 lines in Figure 1 are context lines. The default number of context lines is 3. However, context matching may not be sufficient for function-level patching, since patches may use modified or even new structures and functions.

Therefore, our version analyzer checks for the following properties: 1) context lines match, 2) argument types and return types of functions are the same, 3) The referenced data structures and function signatures are the same. To perform version analysis, we first run `git apply` to apply patches to vulnerable versions. We then parse patched files into an AST and check for these properties. If code changes in vulnerable functions overlap with skipped source ranges, *expression analysis* is performed to ensure that skipped parts do not violate these properties.

If a patch passes feasibility analysis for a version, we consider it as feasible for this version. For example, the `OpenSSL` patch in Figure 1 is feasible for 29 out of 31 vulnerable versions.

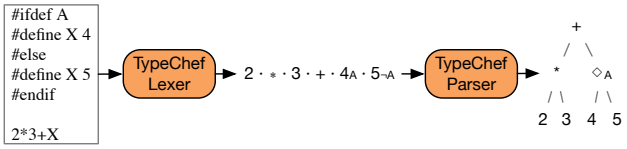


Fig. 5: Variability lexing and parsing using TypeChef.

C. Variability Analysis

Since app developers may use different OSS variants, OSSPATCHER must correctly infer config options that are related to vulnerable functions — to generate correct binary patch using the same config. Although our feasibility analysis (§III-B) can track variability inside vulnerable functions, it cannot reason about variability outside. For example, a function may reference a data structure which contains a field with variable type (i.e., type *int* if macro INT32 is defined, o.w. type *char*). In this case, the value of INT32 is important since it results in different binary layout and offsets.

TypeChef [28] tackles this problem by proposing a variability-aware lexer and parser to build variability-aware AST (VAST). Figure 5 shows the workflow of TypeChef. Nodes in VAST, such as functions, strings, or expressions, are correlated with conditions that enable them. TypeChef has been successfully applied to `OpenSSL` and `Linux` to find type errors in untested config combinations [27, 35]. OSSPATCHER leverages TypeChef to parse OSS into VAST to allow config inference based on app binaries. Nevertheless, TypeChef is not automated and requires manual inputs for its analysis of software, namely separate lists containing platform-dependent headers, open features, and partial configurations, respectively. Platform headers refer to architecture or operating system related macros, such as `__x86_64` and `__linux`. These headers are easy to derive since they are uniquely defined for each platform. Open features include configurable features that developers can choose to enable or disable using `configure` script (e.g., condition `A` in Figure 5) Whereas, partial configuration list contains non-configurable macros with their predefined (fixed) values. Partial configuration must also contain rules to avoid conflicts (e.g., two mutually exclusive macros that cannot be enabled together).

To automatically generate such input lists, we implement pre-analysis steps: *open feature analysis* and *partial config analysis*.

Open Feature Analysis. TypeChef has a different goal; it has been designed to check for incompatible types and developer errors in untested config combinations [27, 35]. To do that TypeChef builds VAST from all source files and enumerates combinations of macro values to check for type errors. In contrast, OSSPATCHER cares only about changed files and included headers and uses VAST for config inference. Since conditional directives are evaluated by the `preprocessor` to selectively enable code blocks, we therefore design a Clang-based analyzer that only performs preprocessing and collects expressions used by conditional directives in source files and include headers. We also recursively collect expressions from skipped code blocks. Collected expressions are then parsed by our *expression analyzer* (§III-B) to extract conditional macros. These macros form the open feature input required by TypeChef.

Partial Config Analysis. Apart from conditional directives, macros are used to set certain OSS attributes, such as timeout or version string. In addition, certain combinations of features are not allowed and may result in a failure of VAST generation since they are syntactically or semantically incorrect. For example, `no-ssl3` is forced if `no-sha` is specified in `OpenSSL`, because SSLv3 uses hashing algorithms internally. TypeChef requires this information to generate VAST. KConfigReader [26] is proposed to extract such information from the Linux kconfig [56] build system. However, this method is not applicable to the GNU build system [74], which is adopted by many OSS projects. Therefore, we build a tool that collects macro definitions and taps into *configure* scripts to extract constraints among features. Although our *partial config analysis* is not complete and may still miss constraints embedded in *Makefile* or other parts of source code, we find the two analyzers greatly reduce the preparation time of TypeChef inputs.

D. Source vs Binary Matching

Patching app binaries at the function level requires locating vulnerable functions, inferring config options, and fixing external references. To achieve these tasks, we design three independent modules: *function matching*, *config inference*, and *variable matching*. Function matching identifies addresses of vulnerable functions and their external function references. Config inference finds out how vulnerable functions are compiled from the source code and generates a config combination for accurate reproduction. Variable matching identifies external variable references in vulnerable functions. Since app binaries are stripped (§II-C), OSSPATCHER should leverage features available in both source code and binaries for source-to-binary comparison. We start with describing the feature extraction process.

Feature Extraction. For source files, we parse their VAST to extract syntactic and semantic features, such as string literals, constants, function calls, and global variable uses along with variability information. We choose simple syntactic and semantic features because, besides being available in compiled binaries, these features are resilient against common compiler optimizations and easy to extract. In contrast, control-flow based features are much harder to define and extract, due to the presence of optional nodes in VAST (i.e. node $\Diamond_A$ in Figure 5).

For binary files, we perform a symbolic summarization of each function present in the binary using an integration of static analysis and symbolic execution based on Angr [70]. Specifically, we conduct a multi-path exploration of each function with the goal of discovering references to a set of predetermined features, including strings, constants, functions, and external variables. Our approach of using per-function symbolic summarization to extract features is quite scalable (more so than whole binary exploration) because our multi-path exploration technique is limited to each function. We do not execute function calls within the function being explored, nor do we execute system or API calls. We just focus on extracting all relevant feature references within one function at a time.

Function Matching. To locate vulnerable functions in libraries, we leverages features from VAST and check if they are present in binaries. When searching for vulnerable functions, we mark them as optional since the corresponding file may not be

compiled, and different parts in these functions can also be optional due to conditional directives. We start matching by first searching for function names in the dynamic symbol table. If names are present, we report matched addresses. Otherwise, we describe candidate functions by reference/call relationship but include optional VAST nodes. We then use Angr to summarize the binary functions and compare with source functions to identify the closest matches. The above algorithm works well if there are abundant syntactic and semantic features in vulnerable functions. To backup this assumption, we show in §V-A that vulnerable functions of most OSS have on average more than 100 lines of code. In addition to vulnerable functions, we also match large functions in binaries to facilitate *config inference*.

Config Inference. To allow reproduction of vulnerable functions with the same configuration, we check for the presence of variability-related features in binaries and solve their conditions to infer values of config options. The key idea is to identify features that are correlated with config options. As shown in Figure 6, we start by collecting config options to be inferred in the VAST of vulnerable functions. Followed by checking if each of these config options are correlated with syntactic and semantic features in the vulnerable functions. For example, option `OPENSSL_NO_EC` is correlated with function reference `OPENSSL_malloc` in Figure 1. Since some options may not enclose syntactic and semantic features in vulnerable functions, we also check other matched large functions during *function matching*. We then check for the presence of these features in binary functions and generate constraints based on the mapping from features to expressions of config options. These constraints are solved using a SMT solver to find the corresponding configuration.

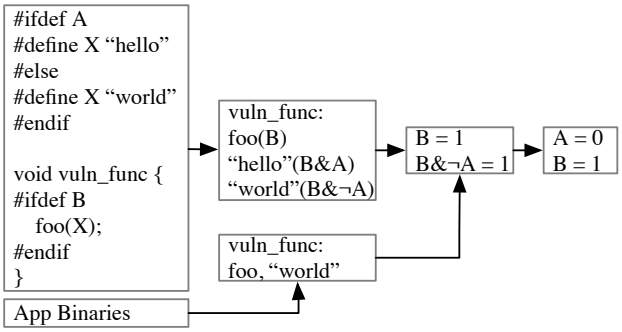


Fig. 6: Explanation of the config inference algorithm.

Variable Matching. Similar to *function matching*, exported variables can be matched by looking up the dynamic symbol table. The case of one hidden variable can also be exclusively matched. However, matching multiple hidden variables becomes challenging due to a lack of enclosed features, which is quite different from matching hidden functions which contain rich features. We solve this challenge by correlating variable references with syntactic and semantic features in the program dependency graph of vulnerable functions. We first compile vulnerable functions into binaries using config options inferred in *config inference*. We then perform foward and backward slicing for external variable references in the program dependency graph to identify features related to each reference in compiled binaries. We consider these features as a description of the variables and use them to match external variable references.

```
1  FITS_FILE *fits_open (const char *filename, ...) {
2  ...
3    if (sizeof (float) == 4)
4    {
5      fits_ieee32_intel = (op32[3] == 0x3f);
6      fits_ieee32_motorola = (op32[0] == 0x3f);
7    }
8    if (sizeof (double) == 8)
9    {
10     fits_ieee64_intel = (op64[7] == 0x3f);
11     fits_ieee64_motorola = (op64[0] == 0x3f);
12   }
```

Fig. 7: An example of static variable usage in GIMP.

For example, Figure 7 shows a function `fits_open` from GIMP, which uses four static variables, including `fits_ieee64_intel` and `fits_ieee64_motorola`. After program slicing and feature extraction, we can identify that each of these variables are associated with different constants. For example, `fits_ieee64_intel` is tied with constant `7` in data flow and `8` in control flow. In contrast, `fits_ieee64_motorola` is tied to `0` and `8`. These features can be used to differentiate these four variables in stripped binaries.

E. Patch Generation and Injection

Once feasible source patches are matched to the app binaries, OSSPATCHER compiles the patched functions and injects them into the vulnerable apps. To make OSSPATCHER practical and portable for various Android systems, we try to minimize runtime overhead and avoid changing the Android system. Inspired by PatchDroid [42], we design OSSPATCHER to perform in-memory patching at the start of app launching. However, we argue that binary rewriting [71, 76] and hot-patching at runtime [3, 8, 73] are also good patching techniques and leave their implementation as future work. To minimize changes to vulnerable apps during patching, OSSPATCHER slices out vulnerable functions into separate files and compiles them into shared libraries. OSSPATCHER then injects the libraries into vulnerable processes and reroutes vulnerable functions to call patched ones in the injected libraries.

Since vulnerable functions can refer to other functions or variables, OSSPATCHER needs to fill in the references to point to the correct memory locations in app binaries. PatchDroid modifies `GOT` entries to reroute vulnerable functions to patched functions and doesn't address the cases where these functions reference the original app binaries, which is not suitable in OSSPATCHER. Normally, external references of libraries are listed as undefined symbols and resolved by the dynamic loader via checking dependent libraries at runtime. However, since vulnerable functions can refer to hidden functions or variables, naive reliance on library dependency does not work. We, therefore, investigate three approaches for fixing external references. The first approach is to modify the dynamic loader to fix external references while loading patch libraries. This design introduces changes to the system, which we try to avoid and incurs overhead during the loading of all libraries. The second approach is to hard-code reference addresses into libraries during compilation. But this requires per-run adaption of libraries due to address space layout randomization (ASLR) [63]. The third approach is to refactor source code to create

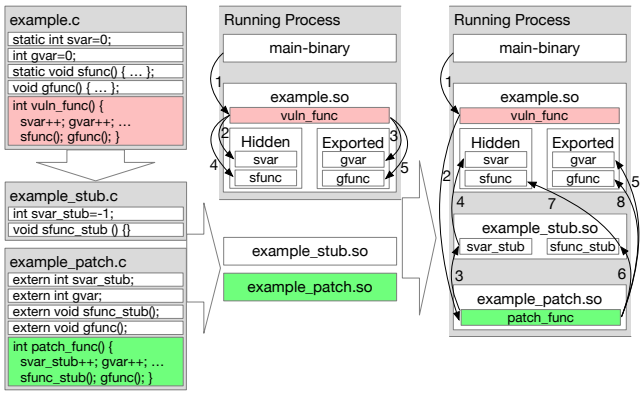


Fig. 8: Workflow of patch generator and patch injector.

stub functions and variables, compile them as dependent stub libraries of patch libraries, and modify stub references to correct locations. OSSPATCHER adopts the stub-based approach, since it avoids changes to the Android system and per-run patch adaption. The workflow of patch generation and injection is explained in Figure 8.

Patch Generation. As shown in Figure 8, OSSPATCHER generates two types of source files which are further compiled into different shared libraries. This design allows OSSPATCHER to fix references in patched functions and point them to correct locations. Given addresses of vulnerable functions and their references identified by `Matcher` (§III-D), we first check the visibility of references by inspecting the dynamic symbol table section (`.dynsym`) in app binaries. For hidden function or variable references, which can be static or non-static, we generate stubs for them and then invoke ClangMove [75], a tool which is capable of moving various definitions, including functions, variables, and classes, from one file into another, to move patched functions into patch files, and stub functions and variables into stub files. We then compile a stub library out of stub files and a patch library out of patch files, with dependencies to the stub library and original vulnerable libraries. Since Android requires a special tool-chain to build binaries for ARM architecture, we perform cross-compilation by setting `CC` or `CXX` to corresponding compilers from the Android NDK [20] (e.g. `arm-linux-androideabi-gcc`). The two libraries comprise our generated patch binaries.

Patch Injection. With generated stub libraries and patch libraries, OSSPATCHER performs in-memory patching at the start of app. When an app is launched in Android, the app forks the `Zygote` process, then loads native libraries through `dlopen` (which internally maps a library into memory using the `open` and `mmap` system calls), and invokes its constructors. OSSPATCHER identifies a time window where patching is most feasible using `ptrace`, which is after the library loads and before library code executes, OSSPATCHER uses `ptrace` to trace the `Zygote` process for its forking of new processes and keeps tracking `open` and `mmap` calls in child processes. Once vulnerable libraries are mapped in memory, OSSPATCHER checkpoints processes using Criu [69] and injects patch libraries with optional stub libraries into them. After injection, external references in patch libraries point to stub libraries or original app binaries. OSSPATCHER then performs detour-based patching to reroute vulnerable functions in app binaries to patched ones in patch libraries and modifies external references of patch libraries to correct locations by overwriting `GOT` entries. Once an app is patched, OSSPATCHER detaches from the target app and the process runs natively with no overhead.

IV. IMPLEMENTATION

OSSPATCHER is written mostly in C++ and Python, with a total of 12K C++ and 15K Python lines of code (LOC). OSSPATCHER builds on several preexisting tools. For example, our data collector is built on cve-search [39] for vulnerabilities and OSSPolice [15] for vulnerable apps. All source analysis and refactoring tools are implemented as independent Clang tools using LibTooling [66]. Variability analysis is build on top of TypeChef [28] and binary analysis is based on Angr [70]. Patch injection uses Criu [69] internally. Here we briefly describe the implementation of each component depicted in Figure 2.

A. Collector

We start by describing our data collection and preparation, including vulnerable OSS and apps that use them.

Vulnerability Database. Numerous efforts have been conducted to discover vulnerabilities and corresponding patches [30, 44, 55]. Out of them, NVD is an accurate collaborative platform for reporting and verifying vulnerabilities manually, and has been used to demonstrate characteristics of security patches [34]. Therefore, OSSPATCHER currently collects vulnerabilities and patches from NVD. However, other vulnerability databases such as Vuddy [30] and OSSFuzz [55] can be incrementally added.

Similar to Li et al. [34], we use cve-search [39] to retrieve CVE information from NVD and look for 40-digit commit hash values in reference links of CVEs. We scanned all 95K CVEs at the time of crawling (Jan, 2018) and identified 5,793 CVEs with at least one commit hash related link. Since OSSPATCHER focuses on patching applications in userspace, we ignore system OSS such as the Linux kernel and uboot. We then clone the remaining 619 OSS and try to checkout corresponding commits, which results in 3,047 valid commits tied to 2,723 CVEs. Out of 3,047 commits, 2,045 are from 307 C/C++ OSS such as `OpenSSL` and 42 are from 22 Java OSS such as `Apache Struts`, implying that the number of documented C/C++ OSS vulnerabilities with patches are much more than Java in NVD. We denote these 307 C/C++ OSS as OSS_{nvd}.

Compile Commands. Clang tools based on LibTooling require compile commands to work, which specifies options such as include directories. These commands can be extracted by adding option `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` to `CMake` [40] or monitoring the compilation process with Bear [43]. Since building each OSS and resolving their OSS dependencies is time-consuming, we leverage OSSFuzz which contains build scripts for a large number of OSS and hook into its build process to get compile commands. OSSFuzz contains 125 OSS at the time of checking (Apr, 2018), denoted as OSS_{fuzz}. We take the intersection of OSS_{nvd} and OSS_{fuzz} and get 39 OSS[2] with 1,111 CVEs and 1,140 patches, denoted as OSS_{eval}. We use OSS_{eval} as our target OSS in evaluation.

[2] Tcpdump is not listed in OSSFuzz but is still included since it is used indirectly by Wireshark which is listed.

Vulnerable Applications. Several studies have been proposed to identify vulnerable Java and C/C++ libraries in apps [4, 15]. Since OSS reuse detection is not the focus of OSSPATCHER, we directly contacted the authors of OSSPolice [15] for a list of flagged vulnerable apps. The obtained list contains 100K unique Android apps tagged with vulnerable OSS versions, denoted as App. Since not all OSS in OSS_{eval} is popularly used by Android apps (e.g. wireshark), we selected 1,000 unique Android apps in total, spanning 10 vulnerable OSS, with 100 apps from each OSS. The 100 apps are randomly selected from apps that use vulnerable versions with feasible patches. We denote the 1,000 apps as App_{eval} and use them as target apps for evaluation.

B. Analyzer

As mentioned in §III-B, our feasibility analyzer contains three independent tools: range analyzer, expression analyzer, and version analyzer. Range analyzer and version analyzer are implemented as `Clang` tools and uses ASTMatcher [65] internally to match and manipulate source code. Expression analyzer includes both frontend AST generation and backend symbolic modeling. We implement the frontend lexer and parser in *expression analyzer* based on Boost Spirit [11] and symbolically represent and solve them using CVC4 [5].

We implement variability analyzer based on TypeChef [28]. Since the current TypeChef requires manual setup, we implement open feature analyzer and partial config analyzer as `Clang` tools to allow semi-automatic setup of TypeChef on new OSS.

C. Matcher

To extract features from binaries, we first identify function addresses using IDA Pro [24]. We start with exported functions, comparing function names preserved in the binary with ones in source files. If vulnerable functions are non-exported, then we go inside every function and extract corresponding features. Specifically, we extract string literals, constants, function calls and number of global variable uses within each function and compare with sources to assist in the identification of non-exported functions. If a vulnerable function is inlined, we currently reject the corresponding patch. However, this can be improved by detecting and patching all functions containing the inlined function. We leave this as future work.

In terms of source-to-binary matching algorithms, we implement function matching ourselves and use the Z3 solver to solve configurations [12] due to its convenient Python interface. To facilitate variable matching, we compile vulnerable functions based on inferred configurations, extract variables using Angr [70], and implement forward and backward slicing for Vex IR [57].

D. Patcher

We implement patch generator as a `Clang` tool, which first generates stubs for hidden references and then invokes Clang-Move [75] to create patch and stub files. We also reuse compile commands from original vulnerable files to compile patch and stub files. Since OSSFuzz has prepared building dependencies, generating shared libraries for different architectures is then achieved by replacing `CC` or `CXX` with compilers of targeted platforms. For example, `arm-linux-androideabi-gcc` from Android NDK [20] is used to generate patches for Android ARM system, and `GCC` is used for Ubuntu X64 system.

To accurately capture the time window of library loading and perform in-memory patching, we implement patch injector as a daemon that monitors forking of the `Zygote` process and tracks when its forked application processes load vulnerable libraries using `ptrace`. Once they are loaded, we use Criu [69] to checkpoint corresponding processes and perform in-memory patching. Once completed, we resume execution and detach from these processes to avoid tracing overhead.

In addition, since address matching and patch generation may suffer from false positives (i.e., a patch which does not perfectly replace the vulnerable code), inspired by PatchDroid [42], we implement a rollback mechanism. When injecting patch libraries, we also inject enter and exit counters at the start/end of patched functions. If a patched app crashes, we catch the crash and check whether the enter and exit counters are the same. If not, we revert the patch and re-execute the function.

V. EVALUATION

In this section, we evaluate the prototype of OSSPATCHER. We start by performing feasibility and variability analysis on 1,140 patches in OSS_{eval}. We then evaluate function matching, config inference, and variable matching algorithms on a labeled dataset. We further apply these algorithms on 1,000 apps in App_{eval}. With the identified configurations and addresses in binaries, we run our patch generator and injector to fix these applications once the vulnerable parts are loaded. We then exercise patched apps using `Monkey` [2] and show that all of them launch and run successfully with negligible memory and performance overhead. To further verify correctness of OSSPATCHER, we collect 10 vulnerabilities with feasible patches and publicly available exploits, including the infamous Heartbleed and Stagefright, and show that all exploits are mitigated.

The evaluation is mainly conducted on a Nexus 5 phone running Android 5.0 (LRX21O) and a Ubuntu 16.04 desktop with 8-core Intel Xeon CPU W3565@3.20GHz and 24GB memory.

A. Code Analysis Statistics

We perform feasibility and variability analysis on the 39 OSS in OSS_{eval}. The analysis shows that 675 out of 1,140 patches are feasible. We selectively show the results for 10 OSS in Table I, since they are used by apps in App_{eval}. Among the columns displayed in Table I, **#CVEs**, **#Patches**, and **#VVs** (vulnerable version) are information collected from NVD and the table is sorted in descending order of **#CVEs**. At the top of the table, `FFmpeg` and `OpenSSL` are reported to have a large number of CVEs, patches, and vulnerable versions, showing that they are the best targets to evaluate OSSPATCHER. In contrast, `Zlib` has only one vulnerable version and may not help in showing cross-version portability of OSSPATCHER.

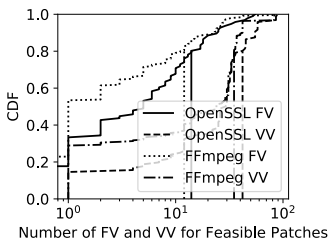
Feasibility Analysis. In Table I, **#FPs** shows the number of feasible patches which is defined as feasible to at least one vulnerable version, **#FVs** shows the number of feasible versions which is defined as being applicable by at least one patch. The two columns show characteristics of patches and capabilities

OSS Name	#CVEs	#Patches	#VVs	#FPs	#FVs	#VFs	#EVFs	$\overline{LOC_{FP}}$	$\overline{LOC_{VF}}$	$\overline{\#Feats_{VF}}$	#MIVFs	#MRVFs
FFmpeg	224	251	156	193	152	197	35	8	102	25	25	30
OpenSSL	89	97	142	80	107	145	105	30	153	31	55	82
FreeType	48	53	49	47	39	64	22	14	105	25	7	15
Libxml2	26	28	79	23	14	117	114	31	219	37	8	22
LibTIFF	20	16	29	13	29	34	32	23	45	7	4	4
OpenJPEG	15	16	2	7	1	13	2	17	112	23	4	10
MuPDF	14	13	10	9	5	18	17	8	83	35	17	33
LibPNG	13	9	577	5	185	8	8	12	109	33	19	42
Curl	4	4	92	4	11	9	4	32	45	9	15	25
Zlib	4	4	1	4	1	4	2	27	143	26	2	4

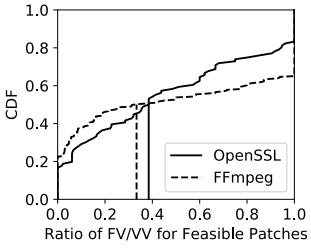
VV: Vulnerable Version, FP: Feasible Patch, FV: Feasible Version, VF: Vulnerable Function, EVF: Exported Vulnerable Function;
$\overline{LOC_{FP}}$: Average Line of Change in Feasible Patch, $\overline{LOC_{VF}}$: Average Line of Code Vulnerable Function;
$\overline{\#Feats_{VF}}$: Average Number of Unique Features in Vulnerable Function;
MIVF: Conditional Macro Inside Vulnerable Function, MRVF: Conditional Macro Related to Vulnerable Function.

TABLE I: Feasibility and variability analysis results of 10 selected OSS.

of OSSPATCHER. For example, 77% of `FFmpeg` and 83% of `OpenSSL` patches are localized and can be automatically applied by OSSPATCHER. Similarly, the fact that 97% of `FFmpeg` and 75% `OpenSSL` vulnerable versions can be patched shows that their code bases are stable and vulnerable functions rarely change across versions until they are fixed. In contrast, 12% of `Curl`'s vulnerable versions can be patched, indicating that `Curl` has made relevant changes to vulnerable functions across versions, which prevents OSSPATCHER from adapting patches across versions. **#VFs** shows the total number of vulnerable functions across all CVEs/patches and **#EVFs** shows exported (non-static) ones among them. $\overline{LOC_{FP}}$ shows average line of change in feasible patches, $\overline{LOC_{VF}}$ shows average size of vulnerable functions[3], and $\overline{\#Feats_{VF}}$ shows average number of unique features in vulnerable functions. From the table, we can see that 197 functions in `FFmpeg` are changed across 193 feasible patches. Similarly, 145 functions in `OpenSSL` are changed among 80 feasible patches. This shows that vulnerabilities can reside in different functions across open source software. $\overline{LOC_{VF}}$ of these two OSS are 102 and 153 respectively, implying that security vulnerabilities are located in medium to large functions. $\overline{\#Feats_{VF}}$ shows that such functions contains a considerable amount of features. In addition, $\overline{LOC_{FP}}$ shows that patches are localized and change only small parts of corresponding vulnerable functions.



(a) CDF of **#FV** and **#VV**.

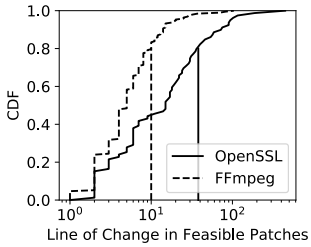


(b) CDF of **#FV/#VV** ratio.

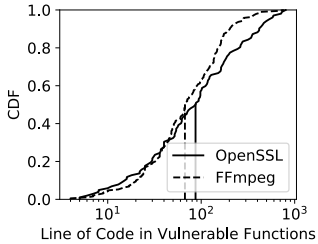
Fig. 9: Cross version analysis for feasible patches.

Apart from overall description of patches, we also present cross version analysis and code size analysis for `OpenSSL` and `FFmpeg` in detail. Figure 9a shows the cumulative distribution function (CDF) of feasible version count and vulnerable version count. It reveals that 80% of patches are tagged with less

[3]The size of a vulnerable function is taken from the latest feasible patch that fixes the particular function.



(a) CDF of line of change in patches.



(b) CDF of line of code in vulnerable functions.

Fig. 10: Code size analysis for feasible patches.

than 40 vulnerable versions and can be applied to less than 15 feasible versions. To understand the capabilities of each patch, we compute the ratio of feasible version count over vulnerable version count (FV/VV) and present its CDF in Figure 9b. The plot shows that 50% of patches have a higher than 35% FV/VV ratio in both `FFmpeg` and `OpenSSL`, implying that OSSPATCHER can be adapted to one third of vulnerable versions for half of the patches. We further inspect release dates of feasible versions versus infeasible versions for patches and find that feasible versions are newer ones while infeasible versions are released years before patch disclosure. This implies that these patches are more likely to be feasible to newer versions of OSS. In addition, to better understand changes in patches and their enclosing functions, we show the CDF of line of change for patches and line of code for vulnerable functions in Figure 10. Figure 10a reveals that 80% of patches changes less than 40 and 10 lines in `OpenSSL` and `FFmpeg` respectively, validating the insight from Li et al. [34] that security patches are localized small in size. Figure 10b shows that 50% of vulnerable functions have more than 90 and 70 lines of code in `OpenSSL` and `FFmpeg` respectively. The decent size of vulnerable functions allows OSSPATCHER to collect a considerable amount of syntactical features for source-to-binary matching. However, it also indicates that patching may incur some memory overhead, since these functions are compiled and injected into running apps.

For infeasible patches, we also investigate them to understand potential improvements to OSSPATCHER. As depicted in §III-B, *feasibility analyzer* analyzes change types, context lines, and reference compatibilities to decide feasibility of a patch. Hence, a patch may be infeasible due to three reasons, namely,

non-functional changes, context mismatches, and incompatible references. Of the 465 infeasible patches, 27% fail due to non-functional changes, 64% do not have matching context lines, and 9% have incompatible references such as new classes or functions with modified signatures. Therefore, we expect that a more comprehensive list of feasible change types and a better mechanism for formatting patches and locating their insertion points (probably with help from OSS developers or security researchers), similar to Coccinelle [47], can further improve the percentage of feasible source patches.

Variability Analysis. In Table I, **#MIVFs** refers to the number of conditional macros used inside vulnerable functions and is collected by *feasibility analyzer*. **#MRVFs** refers to the number of conditional macros related to vulnerable functions and is a superset of **#MIVFs**. In addition to direct conditional macros, **#MRVFs** also considers indirect conditional macros related to data structures or types used by vulnerable functions and is collected by *variability analyzer*. As shown in Table I, different OSS have very different behaviors in terms of variability (i.e. usage of conditional directives). `OpenSSL` uses 55 macros in vulnerable functions, which further expands to 82 in VAST, showing that `OpenSSL` relies heavily on conditional directives and gives users great freedom in customization. In contrast, `FFmpeg` uses fewer macros in vulnerable functions and relies less on conditional directives. We inspected `FFmpeg` source code, which is a large project with many subcomponents, to understand how developers can customize the compilation of the library. Our analysis shows that `FFmpeg` is a highly customizable system but uses a configure script to allow conditional compilation at the module or folder level.

B. Matching Algorithms

To evaluate matching algorithms in `Matcher` when matching source code to binaries, we construct a synthetic dataset for 6 OSS in OSS_{eval} with different OSS variants, due to lack of a labeled dataset. The selected 6 OSS include `Curl`, `FFmpeg`, `LibPNG`, `Libxml2`, `OpenSSL`, and `Wireshark`. We select configurable OSS with a diverse size of code base, ranging from small `Libxml2` to large `Wireshark`, to allow comprehensive evaluation of our proposed algorithms. We obtain the latest versions of them and use a *configure* script to build their variants. `OpenSSL` contains 19 feature related options such as `no-zlib` and `Wireshark` contains 68 such options such as `--enable-dumpcap`. Since enumerating all possible OSS variants is extremely expensive (e.g. 2^{68} for Wireshark), we therefore build a subset of their variants as groundtruth. In particular, we start with the default configuration and specify one feature option at a time to build these OSS (i.e. 1 + 68 for Wireshark). As a result, we get a total of 174 different binaries for the 6 OSS with their debug information such as symbol addresses and config options and use them as groundtruth for evaluating proposed algorithms[4].

Accuracy on Groundtruth. Vulnerable functions for the 6 OSS, as shown in Table I, are relatively large in size and contain a considerable amount of syntactical features for matching. For vulnerable functions which are still available in latest versions, we apply our proposed algorithms to locate them in 174 different binaries of these OSS as well as infer their config options and identify addresses of their external references. By tuning matching threshold numbers, such as ratio of matched features for function equivalence testing (e.g. matching score greater than 0.95 means equivalence), we are able to achieve a precision of 95% at a recall of 82% in source-to-binary matching. We believe the precision and recall are acceptable and further apply these algorithms on real-world binaries.

[4]OSSPATCHER did not need nor have access to this ground truth information.

#Apps	RSA	DSA	ECDSA	DH	ECDH	PSK
1942	✓	✓	✓	✓	✓	✗
315	✓	✓	✓	✗	✗	✗
83	✓	✗	✗	✓	✓	✗

TABLE II: Breakdown of configurations related to function `ssl3_get_key_exchange` for 2,340 Apps using `OpenSSL` 1.0.1e.

To further understand how OSSPATCHER can be improved, we inspect false negatives and false positives in the matched results. As described in §III-D, `Matcher` locates vulnerable functions and their function references, computes config options, and matches variable references. The matching process can fail in each of the three steps, which results in false negatives. For example, function matching can fail because of function inlining (no match) or ambiguous candidates (multiple match), config inference may not be possible due to lack of config-related features, and variable references can be non-distinguishable due to lack of dependent features. Our inspection shows that the three steps introduce 35%, 58%, and 7% of the false negatives respectively, implying that a richer set of features such as control-flow features [16, 17] can help reduce false negatives in source-to-binary matching. We also check false positives and find that they are mainly introduced by compiler optimization such as constant folding or conditional compilation in the presence of ambiguous candidates, which can be improved by more descriptive features as well.

Matching Real-World Applications. Although variability in source code is common as shown in Table I, it is still not clear if app developers adopt non-default setup in practice or not. To understand what real-world apps are doing, we pick all 2,340 apps in *App* that use version 1.0.1e of `OpenSSL`. We run matching algorithms against these apps to infer config options related to a vulnerable function `ssl3_get_key_exchange` and present the breakdown in Table II. Each column in Table II represents a feature that can be optionally excluded using macros, such as `OPENSSL_NO_RSA`, and the first row which excludes only PSK is the default. The results show that 17% of them use non-default configurations, indicating that variability-aware analysis is an essential component in OSSPATCHER.

To further evaluate runtime performance of OSSPATCHER on different OSS, we run matching algorithms on 1,000 apps in App_{eval} and save their identified addresses and configs for further evaluation.

C. Runtime Testing

With the collected addresses and configs for App_{eval}, we run patch generator and injector to fix vulnerabilities in these apps right after the corresponding vulnerable libraries are loaded. Additionally, we run monkey [2] to exercise patched apps for 10 minutes to ensure the normal functioning of

OSS Name	VV	App Name	CVE	EDB	Vulnerable Function	LOC	#GFR	#SFR	#GVR	#SVR
FFmpeg	3.1.2	FFmpeg CLI [33]	2016-10191	[9]	rtmp_packet_read...	117	4	0	0	0
OpenSSL	1.0.1f	Httpd†	2014-0160	32745	dtls1_process_heartbeat	67	5	0	0	0
					tls1_process_heartbeat	66	4	0	0	0
Libxml2	2.9.4	Chrome	2017-15412	[23]	xmlXPathCompOpEval...	197	4	9	0	0
OpenJPEG	2.2.0	PdfViewer [51]‡	2017-15408	[22]	sycc444_to_rgb	39	3	0	0	0
					sycc422_to_rgb	54	4	0	0	0
					sycc420_to_rgb	139	5	0	0	0
					opj_copy_image_header	61	2	0	0	0
					opj_jp2_apply_pclr	131	4	0	0	0
MuPDF	1.11	DocViewer [49]	2017-5991	42138	pdf_run_xobject	162	20	5	0	0
Stagefright	5.1	Hangouts	2015-1538	38124	SampleTable::set...	52	2	0	0	0
BZRTP	1.0.3	Linphone [36]	2016-6271	[64]	bzrtp_packetParser	490	5	0	0	0
OpenLDAP	2.4.42	OpenLDAP†	2015-6908	38145	ber_get_next	204	5	0	0	0
GIMP	2.8.0	GIMP†	2012-3236	19482	fits_decode_header	194	2	3	0	4
Wireshark	2.4.2	Wireshark†	2017-17085	43233	dissect_cip_safety_data	476	16	16	0	30

VV: Vulnerable Version, EDB: Id in Exploit-DB [45] if available, LOC: Line Of Code, GFR: Global Function Reference, SFR: Static Function Reference, GVR: Global Variable Reference, SVR: Static Variable Reference;
† tags Linux applications; ‡ OpenJPEG is used by PDFium, which is further used as PDF rendering library by Android PdfViewer;

TABLE III: Correctness evaluation results for vulnerabilities with public exploits and feasible patches.

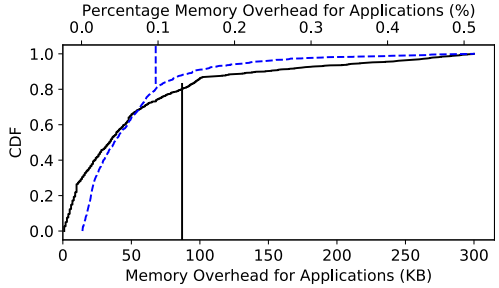


Fig. 11: CDF of memory overhead (KB/Percentage).

these apps. This testing period is practically long enough based on the findings from Choudhary et al [10] that most Android automated testing tools, including monkey, approach their maximum coverage as the testing progresses for 5 to 10 minutes. During our testing, 32% of apps invoked at least one patched vulnerable function. In addition, we record memory and performance overhead introduced by OSSPATCHER. During runtime testing, all patched apps remain functional without any crashes. The memory overhead mainly comes from patch libraries and stub libraries generated by `Patcher`, and as shown in Figure 11, OSSPATCHER incurs less than 80KB (0.1%) memory overhead for 80% of apps. The memory overhead is low because the `Zygote` process consumes roughly 50MB of memory. Since all apps are forked from the `Zygote`, they will consume more memory than it.

In terms of performance overhead, it can be divided into two parts: before-patching (loading) and after-patching (runtime). Our patcher daemon is attached to the `Zygote` process and tracks its forks. Once the vulnerable libraries are loaded, patcher checkpoints the application process, performs in-memory patching, and detaches upon finishing. During our testing, all the apps load vulnerable libraries upon start, and patcher incurs a loading delay of less than 350 milliseconds for 80% of apps. As for runtime overhead, since apps are patched natively using shared libraries, run with normal input (i.e. no crafted files to crash enclosed `FFmpeg` libraries), and remain functional during testing, we observe almost no delay in terms of responsiveness. Since we use detour-based patching, the runtime overhead is only the trampoline instructions. Therefore, similar to other works (e.g. PatchDroid [42]), we empirically conclude that the runtime overhead is negligible.

D. Exploitation and Correctness Verification

In order to verify the correctness of OSSPATCHER, it would be ideal to attack a patched app with a previously working exploit to check whether the exploit is stopped or not. Since apps in App_{eval} are closed-source and automatically generating exploits for them based on vulnerabilities is an orthogonal direction, we verify correctness of OSSPATCHER using apps with publicly available exploits, presented in Table III. We include the infamous Heartbleed and Stagefright as well as a recent exploit for Android Chrome (`CVE-2017-15412`). In addition to vulnerability and exploit information, we also present details of vulnerable functions in Table III, to show the size of vulnerable functions and their function references and variable references. We verify OSSPATCHER using 6 Android apps and 4 Linux apps, to show the capability of OSSPATCHER in patching both Android apps and other Linux-based apps. For collected exploits, we start by validating that they work on vulnerable versions and are blocked in newer (fixed) versions of OSS libraries. We then run OSSPATCHER to compile source patches into shared libraries and patch them into the vulnerable apps. Our evaluation shows that all of these exploits are successfully stopped by OSSPATCHER. We discuss three representative cases below.

Android Chrome. As a large open source project, Chrome reuses many other OSS as well, such as `FFmpeg`, `Libxml2` and `WebRTC`. On Android, Chrome compiles them into a giant library, named `crazy.libchrome.so`, and uses a wrapper to interact with these functionalities. To improve the security of Chrome, Google launches bug bounty programs to encourage security researchers to test and submit vulnerabilities with exploits, among which we identified `CVE-2017-15412` that exploits `Libxml2` in Chrome using a crafted xml file [23]. We then use OSSPATCHER to patch function `xmlXPathCompOpEvalPositionalPredicate` by locating necessary addresses and inferring its compilation options. After injecting this patch, we found the exploit to be effectively thwarted.

Stagefright. Libstagefright is one of Android's built-in system libraries and is used by system services as well as first-party apps, such as Hangouts, to process multimedia files. There are several exploits available for the infamous stagefright bug, and we demonstrate OSSPATCHER using exploit 38124 in Exploit-DB [45] which crafts a malicious mp4 file. We use a Nexus 5 phone running Android 5.0, which is subject to this vulnerability to carry out the experiment. Before patching, the exploit can start a reverse shell through Hangouts. After patching `SampleTable::setSampleToChunkParams` using OSSPATCHER, the exploit is stopped.

Heartbleed. Apache web server, Httpd, uses `OpenSSL` for hosting websites over https. Heartbleed vulnerability allows attackers to peek server's memory. We setup Httpd with 1.0.1f version of `OpenSSL` in a docker container and turn on https. With exploit 32745, we are able to dump memory of web server. After patching the server daemons at runtime by fixing function `dtls1_process_heartbeat` and `tls1_process_heartbeat`, an attacker cannot exploit the server any longer.

VI. DISCUSSION

A. Patching Techniques

We demonstrate OSSPATCHER with live-patching at the start of app launching. But OSSPATCHER can be adapted to perform hot-patching at runtime which has benefits such as continual service. The difference between these is timing of injection and requirements on patches. Hot-patching needs to ensure that vulnerable functions are not being executed and patches are stateless. However, both approaches require root privilege due to the use of `ptrace` and may increase attack surface if adopted by users. On the contrary, binary rewriting doesn't require root privilege and minimizes attack surface. While prototyping OSSPATCHER, we could have chosen either, however, in-memory patching allows us to safely revert the patch on exception and helps in debugging. We leave implementation of other patching techniques and their comparison as future work.

B. Alternative Deployments

While this paper focuses on patching Android apps, the techniques used by OSSPATCHER can also be applied to patch vulnerabilities in userspace programs of any Linux-based system, particularly apps on Docker Hub, of which more than 80% of official apps have been reported to have at least one highly severe vulnerability [58, 62]. In fact, the correctness verification of 4 Linux apps (e.g. Httpd) in Table III is performed using Docker for better reproducibility.

OSSPATCHER is a system to help third-parties patch public vulnerabilities in applications for the sake of users, which assumes unavailability of source code. However, we argue that OSSPATCHER can also be adapted to help app developers to push their security patches quickly to users.

C. Limitations

Information Authenticity. OSSPATCHER assumes that the information in NVD is accurate. But this assumption may not be true. For example, `CVE-2016-10156` is a vulnerability correlated with `systemd` which allows privilege escalation. The description mentions that version 228 is vulnerable and 229 fixes the problem. The CVE entry has two commit links in the reference section. We tested the corresponding 41171 exploit in Exploit-DB for this CVE. We found it working on version 228 through 236 and was stopped in 237, which shows that the claim made in the description is not correct. Moreover, one of the two commits is already included in version 228, indicating that developers may have back-ported the commit, or the commit is not a patch. However, we argue that checking authenticity of information is orthogonal to OSSPATCHER and can be addressed by other approaches such as manual reviews and regression testing.

System Capability. OSSPATCHER currently classifies limited types of changes as feasible and supports VAST building for only C language due to limitation of TypeChef. However, OSSPATCHER can be extended to support other types of changes that result in localized binary changes. For example, patch `188eb6b` of `LibPNG` is considered infeasible by OSSPATCHER due to its change of *typedef* statements. However, it can be classified as feasible by a more complex analysis which is capable of identifying and separating functional changes versus non-functional changes, such as version string update. Similarly, OSSPATCHER can support C++, by rewriting TypeChef as a clang tool. In this paper, we avoid this engineering overhead and prioritize demonstration of practicality while prototyping OSSPATCHER. But we argue that future research can be conducted to clearly define feasible patches and identify challenges in VAST building for C++.

Dynamic Code Coverage. During our testing, we found that many of the patches applied to *deep vulnerabilities* — often beyond the reach of symbolic/dynamic analysis due to precise environmental requirements. Our automated dynamic analysis in §V was only able to exercise patched vulnerable functions in 32% of the tested apps. This led us to manually verify the patches in §V-D and motivated the design of our automated rollback mechanism presented in §IV-D. As dynamic code coverage techniques advance, we will continue to improve the automated verification of our patches.

VII. RELATED WORK

Previous efforts related to OSSPATCHER can be categorized into three lines of work.

A. Automatic Patching

Researchers have proposed approaches to automatically generate patches by learning from previous patches. For example, CodePhage [60] and CodeCarbonCopy [59] patches buggy apps by borrowing code from fixed donor apps. Prophet [38] automatically generates patches from successful OSS patches and assigns candidate patches with probabilistic scores. Due to the fact that security vulnerabilities are localized and have fixed types, researchers also proposed systems to patch binaries directly based on domain knowledge or machine learning techniques. For example, ClearView [48] patches errors based on execution failures. Axis [37] and AFix [25] focus on automatically fixing atomicity violations. Schulte et al. [52–54] propose evolutionary algorithms to repair programs. GenProg [32] and Par [29] propose genetic-programming methods for patching.

BinSurgeon [18] and AutoFix-E [72] allow users to write patches using templates or source annotations. These works propose various ways to fix programs under the assumption that patches are not available. We consider these works as orthogonal to OSSPATCHER which works on existing patches.

Researchers have also proposed patching techniques under the assumption that patches are available. For example, Ksplice [3] and Kpatch [73] performs Linux kernel live patching based on existing kernel patches. Karma [8] performs Android kernel patching based on manually written lua patches. InstaGuard [7] presents a new method for applying patches leveraging ARM debugging primitives. BISTRO [13] proposes techniques to extract binary component and embed them into other binaries. There are also works that focuses on patching native libraries (C/C++) or Dalvik binaries (Java) Android apps [14, 42, 76]. However, these works either assume availability of source code and config options of compilation [3, 7, 8, 73] or assume impractical availability of binary patches [13, 42]. In contrast, OSSPATCHER assumes that apps are closed source and source patches of OSS are available, which we believe is a practical assumption for Android apps and OSS. Compared to patching Java libraries [14, 76] in Android apps, OSSPATCHER focuses on patching native libraries written in C/C++, which are reported to be present in 40% of all apps [1], and have more CVE entries as well as unique challenges.

B. N-day Vulnerability Detection

Various approaches have been proposed to identify known (n-day) vulnerabilities in binaries at different granularities. LibScout [4] and OSSPolice [15] detect libraries and correlate them with existing vulnerability data to identify vulnerable ones. OSSPATCHER reuses them to identify apps with vulnerable OSS versions.

Discovre [16], Genius [17], and Gemini [77] compile vulnerable functions into binaries and directly search for them in firmware images. Fiber [78] proposes fine-grained patch presence test to allow more accurate bug finding. However, these approaches do not consider OSS variants and may need to compile and search an exponential number of binaries. To overcome this limitation, OSSPATCHER proposes variability-aware matching algorithms to identify vulnerable functions as well as config options and reference addresses.

C. Variability-aware Code Analysis

There has been work that carry out variability-aware static analysis of large and complex software systems, such as the Linux kernel, Busybox, etc. to detect compile time bugs, dead code, inconsistent configuration, etc. Undertaker [6] is a suite of tools to carry out variability-aware static analysis of Linux kernel source code for dead code and related bugs introduced by C preprocessor directives. Vampyr [80] is a part of the Undertaker suite that performs variability-aware static coverage analysis of kernel drivers. KConfigReader [26] uses Undertaker to analyze a Linux kernel variability model (kconfig files) and translate it into a propositional formula for automated reasoning with SAT solvers. The TypeChef [28] tool also does variability-aware static analysis of software systems to detect compile and link time errors introduced by the C preprocessor. They introduce an AST with choice nodes to encode variability information. These works all focus on analyzing source code for problems related to variability.

In contrast, OSSPATCHER focuses on bridging the gap between OSS variants and their compiled counterparts in binaries. OSSPATCHER reuses TypeChef to generate VAST and performs source-to-binary matching for subsequent patching operations.

VIII. CONCLUSION

In this paper, we presented OSSPATCHER, the first automated system that fixes n-day OSS vulnerabilities in app binaries by automatically converting feasible source patches into binaries and performing in-memory patching. We focus on fixing uses of vulnerable OSS written in C/C++ for Android apps while prototyping OSSPATCHER. We populated OSSPATCHER with 39 OSS and 1,000 Android vulnerable apps. Our evaluation shows that 675 source patches are feasible and OSSPATCHER fixes vulnerabilities with negligible memory and performance overhead. We, therefore, conclude that OSSPATCHER is a practical system and can be deployed by vendors or end users to fix n-day vulnerabilities without involving app developers.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their constructive comments and feedback. We also thank Professor Thorsten Holz for his guidance while shepherding this paper. This research was supported by ONR under grants N0001409-1-1042, N00014-15-1-2162, and N00014-17-1-2895, and the DARPA Transparent Computing program under contract DARPA-15-15-TCFP-006, and NSF under Award 1755721. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[2] Android Developers, "Monkey runner," 2018. [Online]. Available: https://developer.android.com/studio/test/monkeyrunner/

[3] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*, Nuremberg, Germany, Mar. 2009.

[4] M. Backes, S. Bugiel, and E. Derr, "Reliable Third-Party Library Detection in Android and its Security Applications," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[5] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, Snowbird, UT, Jul. 2011.

[6] CADOS Developers, "The undertaker is an implementation of our preprocessor and configuration analysis approaches," 2018. [Online]. Available: https://undertaker.cs.fau.de

[7] Y. Chen, Y. Li, L. Lu, Y.-H. Lin, H. Vijayakumar, Z. Wnag, and X. Ou, "Instaguard: Instantly deployable hot-patches for vulnerable system programs on android," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[8] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, CANADA, Aug. 2017.

[9] P. Cher, "ffmpeg remote exploitaion results code execution," 2016. [Online]. Available: http://www.openwall.com/lists/oss-security/2017/02/02/1

[10] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lincoln, Nebraska, Sep. 2015.

[11] J. de Guzman and H. Kaiser, "The boost spirit library," 2016. [Online]. Available: http://boost-spirit.com/home/

[12] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[13] Z. Deng, X. Zhang, and D. Xu, "Bistro: Binary component extraction and embedding for software security applications," in *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS)*, Egham, United Kingdom, Sep. 2013.

[14] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, Texas, Oct. 2017.

[15] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, Texas, Oct. 2017.

[16] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovre: Efficient cross-architecture identification of bugs in binary code," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[17] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[18] S. E. Friedman and D. J. Musliner, "Automatically repairing stripped executables with cfg microsurgery," in *Proceedings of the 15th IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASO)*, Cambridge, MA, Sep. 2015.

[19] GCC Administrator, "The c preprocessor," 2018. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-2.95.3/cpp_1.html

[20] Google Inc., "Android Studio, The Official IDE for Android," 2016. [Online]. Available: https://developer.android.com/studio/index.html

[21] ——, "App Security Improvement Program," 2016. [Online]. Available: https://developer.android.com/google/play/asi.html

[22] ——, "Pdfium heap buffer overflow vulnerability in openjpeg," 2017. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=762374

[23] ——, "Uaf/double free with xslt xpath expressions containing function calls in predicates," 2017. [Online]. Available: https://bugs.chromium.org/p/chromium/issues/detail?id=727039

[24] Hex-Rays SA, "Ida is the interactive disassembler: the world's smartest and most feature-full disassembler," 2018. [Online]. Available: https://www.hex-rays.com/products/ida/

[25] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *ACM Sigplan Notices*, vol. 46, no. 6. ACM, 2011, pp. 389–400.

[26] C. Kästner, "Differential testing for variational analyses: Experience from developing kconfigreader," *arXiv preprint arXiv:1706.09357*, 2017.

[27] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger, "Variability-aware parsing in the presence of lexical macros and conditional compilation," in *Proceedings of the 22th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, Oct. 2011.

[28] A. Kenner, C. Kästner, S. Haase, and T. Leich, "Typechef: toward type checking# ifdef variability in c," in *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development (FOSD)*, Eindhoven, Netherlands, Oct. 2010.

[29] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, May 2013.

[30] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[31] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society, 2004, p. 75.

[32] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog-a generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, p. 54, 2012.

[33] M. Len, "Ffmpeg cli," 2018. [Online]. Available: https://play.google.com/store/apps/details?id=org.magiclen.ffmpeg.cli

[34] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, Texas, Oct. 2017.

[35] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer, "Scalable analysis of variable software," in *Proceedings of the 18th European Software Engineering Conference (ESEC) / 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Saint Petersburg, Russia, Aug. 2013.

[36] Linphone Developers, "Linphone, for smartphones, tablets and mobile devices," 7 2018. [Online]. Available: http://www.linphone.org

[37] P. Liu and C. Zhang, "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, Zurich, Switzerland, Jun. 2012.

[38] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *ACM SIGPLAN Notices*, vol. 51, no. 1. ACM, 2016, pp. 298–312.

[39] J. Long, "cve-search - a tool to perform local searches for known vulnerabilities," 2018. [Online]. Available: http://cve-search.github.io/cve-search/

[40] K. Martin and B. Hoffman, *Mastering CMake: a cross-platform build system*. Kitware, 2010.

[41] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca, "Discipline matters: Refactoring of preprocessor directives in the# ifdef hell," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 453–469, 2018.

[42] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: Scalable third-party security patches for android devices," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2013.

[43] L. Nagy, "Bear is a tool that generates a compilation database for clang tooling." 2018. [Online]. Available: https://github.com/rizsotto/Bear

[44] National Institute of Standards and Technology, "National vulnerability database," 2013. [Online]. Available: http://nvd.nist.gov/

[45] Offensive Security, "Exploit database archive," 2018. [Online]. Available: https://www.exploit-db.com/

[46] openSUSE contributors, "zypper-docker: Easy patch and update solution for docker images," 2018. [Online]. Available: https://software.opensuse.org/package/zypper-docker

[47] Y. Padioleau, R. R. Hansen, J. L. Lawall, and G. Muller, "Semantic patches for documenting and automating collateral evolutions in linux device drivers," in *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*. ACM, 2006, p. 10.

[48] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[49] Privacy Apps, "Document viewer," 1 2018. [Online]. Available: https://play.google.com/store/apps/details?id=org.sufficientlysecure.viewer

[50] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code." in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[51] B. Schiller, "Android pdfviewer," 2018. [Online]. Available: https://github.com/barteksc/AndroidPdfViewer

[52] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, "Automated repair of binary and assembly programs for cooperating embedded devices," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[53] E. Schulte, S. Forrest, and W. Weimer, "Automated program repair through the evolution of assembly code," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Antwerp, Belgium, Sep. 2010.

[54] E. M. Schulte, W. Weimer, and S. Forrest, "Repairing cots router firmware without access to source code or test suites: A case study in evolutionary software repair," in *Proceedings of the 24th Annual Conference on Genetic and Evolutionary Computation (GEC)*, Madrid, Spain, Jul. 2015.

[55] K. Serebryany, "Oss-fuzz: Google continuous fuzzing service for open source software," *USENIX Association*, 2017.
[56] S. She and T. Berger, "Formal semantics of the kconfig language," *Technical note, University of Waterloo*, vol. 24, 2010.
[57] Y. Shoshitaishvili, "Python bindings for valgrinds vex ir," 2014.
[58] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the 7th Annual ACM Conference on Data and Applications Security and Privacy (CODASPY)*, Scottsdale, AZ, Mar. 2017.
[59] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy," in *Proceedings of the 25th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Paderborn, Germany, Sep. 2017.
[60] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Portland, OR, Jun. 2015.
[61] R. Stallman, "Using the gnu compiler collection: a gnu manual for gcc version 4.3. 3," *CreateSpace, Paramount, CA*, 2009.
[62] B. Tak, C. Isci, S. Duri, N. Bila, S. Nadgowda, and J. Doran, "Understanding security implications of using containers in the cloud," in *USENIX Annual Technical Conference (USENIX ATC'17)*, 2017.
[63] P. Team, "Pax address space layout randomization (aslr)," *Phrack, March*, 2003.
[64] G. Teissier, "Proof of concept for zrtp man-in-the-middle," 7 2017. [Online]. Available: https://github.com/gteissier/CVE-2016-6271
[65] The Clang Team, "Ast matcher reference," 2018. [Online]. Available: http://clang.llvm.org/docs/LibASTMatchersReference.html
[66] ——, "Libtooling is a library to support writing standalone tools based on clang," 2018. [Online]. Available: https://clang.llvm.org/docs/LibTooling.html
[67] L. Torvalds and J. Hamano, "Git: Fast version control system," 2010. [Online]. Available: http://git-scm.com
[68] G. van Rossum, "Unified diff format," 2018. [Online]. Available: https://www.artima.com/weblogs/viewpost.jsp?thread=164293
[69] Virtuozzo, "Checkpoint/restore in userspace." [Online]. Available: https://www.criu.org
[70] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *Cybersecurity Development (SecDev), 2017 IEEE*. IEEE, 2017, pp. 8–9.
[71] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.
[72] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, Trento, Italy, Jul. 2010.
[73] Wikipedia contributors, "Kpatch — Wikipedia, the free encyclopedia," 2017. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Kpatch&oldid=798558878
[74] ——, "Gnu build system — Wikipedia, the free encyclopedia," 2018. [Online]. Available: https://en.wikipedia.org/w/index.php?title=GNU_Build_System&oldid=821748569
[75] H. Wu, "A prototype tool for moving class definition to new file," 2018. [Online]. Available: https://github.com/llvm-mirror/clang-tools-extra/tree/master/clang-move
[76] J. Xie, X. Fu, X. Du, B. Luo, and M. Guizani, "Autopatchdroid: A framework for patching inter-app vulnerabilities in android application," in *Proceedings of the IEEE International Conference on Communications (ICC)*, Paris, France, May 2017.
[77] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, Texas, Oct. 2017.
[78] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.
[79] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu, "Embroidery: Patching vulnerable binary code of fragmentized android devices," in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 47–57.
[80] A. Ziegler, V. Rothberg, and D. Lohmann, "Analyzing the impact of feature changes in linux," in *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems*. ACM, 2016, pp. 25–32.