Binary Quilting to Generate Patched Executables without Compilation

Anthony Saieva
Columbia University
ant@cs.columbia.edu

Gail Kaiser
Columbia University
kaiser@cs.columbia.edu

ABSTRACT

When applying patches, or dealing with legacy software, users are often reluctant to change the production executables for fear of unwanted side effects. This results in many active systems running vulnerable or buggy code even though the problems have already been identified and resolved by developers. Furthermore when dealing with old or proprietary software, users can't view or compile source code so any attempts to change the application after distribution requires binary level manipulation. We present a new technique we call *binary quilting* that allows users to apply the designated minimum patch that preserves core semantics without fear of unwanted side effects introduced either by the build process or by additional code changes. Unlike hot patching, binary quilting is a one-time procedure that creates an entirely new reusable binary. Our case studies show the efficacy of this technique on real software in real patching scenarios.

KEYWORDS

binary analysis; binary patching

ACM Reference Format:
Anthony Saieva and Gail Kaiser. 2020. Binary Quilting to Generate Patched Executables without Compilation. In *2020 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '20), November 13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3411502.3418424

1 INTRODUCTION

In most cases the typical software maintenance lifecycle is sufficient to keep code updated and secure. Users report bugs, developers investigate the issue, resolve the problem, release a new version of the software in line with the pre-existing release schedule, and users deploy the new version of the software. However, in software that deals with critical system functionality users tend to update software only when absolutely necessary for fear of updates introducing side effects that disrupt service. Since most software is released according to a common schedule, new releases often contain many modifications most of which a singular user would deem unnecessary. These pose an unnecessary risk and in fact the user may not be able to integrate new versions if relevant interfaces have been replaced or wiped away. This leaves many users in an awkward position: they have code with known deficiencies and the corresponding updates, but they also can't apply those updates.

While not necessarily problematic for standard feature releases, this situation proves disastrous when failing to fix security vulnerabilities. In the Equifax hack of 2017 [21], attackers exploited a well-known vulnerability in the Apache Struts library that had been fixed months earlier, but had not been applied to Equifax's production code, to steal approximately 145.5 million U.S. consumers' personal data, including their full names, Social Security numbers, birth dates, addresses, and driver license numbers.

If users had access to the version control repository and build process, they could theoretically search the change log to build the specific version that suits their needs, but proprietary and legacy code users either don't have access or no such build exists since many unrelated changes have been included in the update. Instead we present a new technique called *binary quilting* that allows users to apply the minimum patch designated by the developers as fixing *only* the targeted software bug. This eliminates side effects from unwanted changes, thus providing a way for users to apply updates to legacy binaries while still supporting necessary functionality.

We developed a technique that integrates with the build process called Binary Patch Decomposition (BPD) that automatically creates the necessary metadata for the quilting procedure (BinQuilt). BPD creates metadata associated with each commit that might be of interest to users. The developers then supply this metadata for each bug-fix patch along with the new release (potentially consisting of many commits), to enable BinQuiltto operate in the user environment (without developer source code or build process).

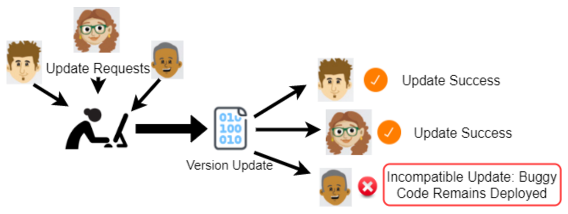
This work presents the following contributions:

(1) *Binary Quilting* (BinQuilt) - a technique leveraging the Egalito binary "recompilation" framework [25] to generate patched binaries by combining executables. To try to minimize confusion, we henceforth refer to Egalito's notion of "recompiling", from its intermediate representation without source code or a conventional compiler, as "(re)generation".
(2) *Binary Patch Decomposition* (BPD) - a build process integrated technique to map specific source code changes to the corresponding binary changes.
(3) Peanut (Patch dEcomposition ANd qUilTing), an implementation of BPD and BinQuilt for Linux on x86.
(4) Case studies demonstrating Peanut's potential in realistic update scenarios.

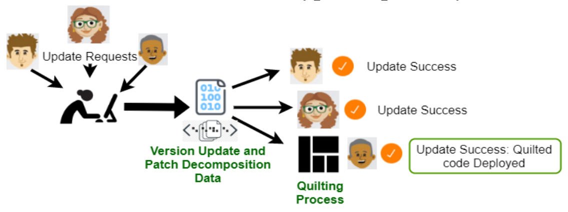
2 BACKGROUND AND RELATED WORK

2.1 ELF Format and Existing Binary Diffs

ELF files are the standard executable format on Linux operating systems, which contain all the metadata required to link, load,



(a) Typical Update Cycle



Update Requests
Version Update and Patch Decomposition Data
Quilting Process
Update Success
Update Success
Update Success: Quilted code Deployed

(b) Update Cycle with Quilting

Figure 1: Problem Overview

and run the program. This extra information includes the *symbol table* which maps the location of every symbol in the binary to the name of the symbol in the associated *string table*. The string table is simply a list of strings delimited by null bytes. This data structure allows for linking across object files and executable ELF files. The ELF file also contains a *relocation table* for link time and load time symbol resolutions. In many production binaries symbols are stripped for performance and security so may not be available for analysis. Lastly, the ELF format splits the binary into data and code segments with different permissions for security reasons.

Binary diffs is a well studied problem, however it is usually taken from the perspective of a reverse engineer. Such tools include BinDiff [26], BinHunt [13], iBinHunt [20], BinSlayer [11], BinSequence [17], and SemDiff [24]. All of these tools rely on some variant of control flow graph extraction and comparison. Usually this entails significant computing overhead and heavy analysis. However in the context of binary quilting, these techniques fail to leverage information available during the build process, so these diffs may pick up on changes not relevant to the patch desired by a given user, resulting in an inaccurate quilting effort.

2.2 x86 Calling Conventions

While the minimum binary diff information provided by BPD relays where changes occur, since small changes in source code can result in large semantic changes like changing register usage and memory layout, copying semantically equivalent or nearly equivalent code from one binary to another is precarious. Fortunately the Intel x86 architecture defines strict calling conventions as to what registers must be maintained by the calling and callee functions as well as the state of the stack for arguments to be passed from one function to another [12]. As long as the arguments are consistent between two function versions, the initial state of the stack and the register layout is presumed to be the same. In the event a function's arguments change, this guarantee no longer holds true, but the calling functions must also change to accommodate the new signature so for the purposes of applying patches, these calling functions provide the points of interface between original and new versions.

2.3 Related Work in Hot Patching and Binary Rewriting

Binary rewriting has been used for many reasons including implementing defenses, automatic program repair, hot patching, and optimization. Hot patching is an interesting example since it requires conserving dynamic program state at the time the repair is applied similar to binary quilting. Katana [22] has highly sophisticated mechanisms for handling this problem, many of which would augment our current quilting procedure, but relies on trampolines to apply the patches which could incur significant overhead the same as [18]. Other binary rewriting mechanisms like Zipr [15, 16] raise the binary to a higher level IR which allows for increased efficiency in the reassembly process similar to Egalito [25] but have demonstrated generic binary level defense transformations instead of semantically complex bug specific patching.

3 DESIGN

3.1 Binary Patch Decomposition

3.1.1 Source code diff analysis. Standard version control systems (VCS) like Git [14] use source code diffs to track source code changes. For any given code change the VCS maintains a record of exactly what changed. Most VCS's save this record following a standard patchfile convention as shown in Figure 2. This patchfile includes the modified function name, line numbers, source code changes, and file names among other information.

3.1.2 Symbol Table Parsing. BPD parses the patchfile, and extracts the names of the modified function(s). In the event that the change modifies a data related symbol like a global variable or a string then that symbol is marked as modified. After the executable is built Peanut's BPD parses the symbol table to extract the location of the modified function in the new binary. The symbol name provides an interface between the source code changes and the binary addresses as shown in Figure 3 where the addresses are denoted in hex on the left and the symbols in text on the right. The symbol table also includes a description of the data type. In the example shown the symbols are function names designated by FUNC but if they were data symbols like global variables or structs they would be designated `OBJECT`. This information is relevant because depending on the data type they will be in different sections of the binary. The symbol table also specifies the size of the symbols so we can extract the length of the functions and the size of the data types. Once we know the location, size and contents of the modified, even if the symbol table is stripped after building (usually pre-deployment) we can still identify the patched portions of the binary completely.

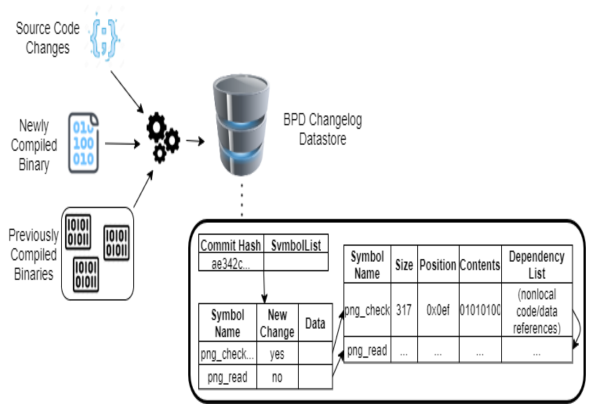
3.1.3 Database storage for on the fly patch decomposition. BPD is designed to track binary changes without interfering with the normal development process. One way to track changes would be to keep a separate copy of the binary for each commit and then extract information as needed to create the binary patch. However keeping so many versions of the binary doesn't really make sense since most of the information is redundant. BPD stores only the necessary information to construct the binary patch as shown in

```
 ...
@@ -3167,10 +3167,13 @@ png_check_chunk_length(...) {
   ...
- (png_ptr->width * png_ptr->channels  // source changes
   ...
+     (size_t)png_ptr->width
+     * (size_t)png_ptr->channels
```

Figure 2: libpng-bug-1 Abbreviated Example Patchfile

```
0000000000003fe0    56 FUNC ... png_check_chunk_name
0000000000004020   221 FUNC ... png_check_chunk_length
0000000000004100   172 FUNC ... png_read_chunk_header
```

Figure 3: libpng-bug-1 Abbreviated Symbol Table Entries



Commit Hash	SymbolList
ae342c...	

Symbol Name	New Change	Data
png_check...	yes	
png_read	no	

Symbol Name	Size	Position	Contents	Dependency List
png_check	317	0x0ef	01010100	(nonlocal code/data references)
png_read	...	...	...	...

Figure 4: Datastore Implementation

Figure 4. Essentially the binary is split based on it's symbols and the symbol contents and meta data are stored in the database. The database must reflect the binary's structural information to apply the patch after symbols have been stripped, its contents so the patch can be built, and maintain flexibility so a patch can be constructed from multiple code versions. This means that each piece of code must keep track of its external references across versions.

3.1.4 Metadata Construction Algorithm. When a user asks for a particular patch, Peanut's BPD must transform the entries in the database into metadata that can be leveraged to build and apply the patch in the user environment. The process to construct this metadata is described in Algorithm 1. The algorithm takes the parsed original and new binaries as inputs, and then for every changed symbol it has to do two things. First, if the symbol exists in the old binary, it must add the old size and position data so any references to this piece can be removed. Second, BPD must search through all the dependencies of each changed symbol (a dependency is any nonlocal reference); if the dependency existed in the old binary (as per the symbol name), then the metadata can simply add the new code piece and the location of its own dependencies. If the dependency doesn't exist, it must be added to the metadata as a newly changed symbol and its dependencies searched as well.

Algorithm 1: Pseudocode: Metadata Construction

```
Result: Metadata(old_info, new_info)
Input  : parsed original binary OV; parsed new binary NV;
         BPD datastore DB
Output : metadata information required to construct patch
         MD
getMetadata ($OV, NV, DB$)
    original_code = DB.get_code_pieces(OV);
    new_code = DB.get_code_pieces(NV);
    res.new_info ← ∅, res.old_info ← ∅;
    changed_symbols = DB.getChangedSymbols(NV);
    foreach $symbol \in changed\_symbols$ do
        res.new_info.add(symbol);
        if $symbol \in original\_code$ then
            res.old_info.add(symbol);
            foreach $cp \in symbol.dependency\_list$ do
                if $cp \notin original\_code$ then
                    sym = Symbol(cp, newChange=True);
                    res.new_code.add(sym);
            end
    end
    return Metadata(res.new_info, res.old_info)
```

Algorithm 2: Pseudocode: Binary Quilting

```
Result: Updated Pointers
Input  : parsed original binary OV; parsed new binary NV;
         BPD metadata MD
Output : The quilted binary
getQuilted ($OV, NV, MD$)
    foreach $ref \in OV$ do
        if $isOverwritten(ref.target)$ then
            ref.target = metadata.new_info[target_symbol];
    end
    foreach $ref \in metadata.ne\_info$ do
        if $isCodeRef(ref.target)$ then
            //adjust pointer
        if $isDataRef(ref.target)$ then
            if $OV.dataSection(ref)$ then
                //adjust pointer
            else
                //add new data
                //adjust pointer
            end
        if $isPLTRef(ref.target)$ then
            //update PLT entries
    end
```

3.2 Binary Quilting Procedure

Once the BPD metadata has been produced and shipped to the user as described in the previous section, now the user must run Peanut's quilting function, BinQuilt, to actually create the patched binary and the fully linked form of the patch. The fully quilted binary is shown in Figure 5.

In order to parse the binaries and identify the reference locations as well as the regeneration phase we rely on the Egalito framework. It lifts the binary into an intermediary representation (EIR) that extracts the control flow graph and provides an interface to the data sections. Egalito also provides a portable XML format (HOBBIT) that encodes all the structural and data related information. Egalito then generates an ELF file to run on the specified hardware. This provides some interesting opportunities for updating legacy binaries to different hardware.

In order to quilt the new version of the binary into the old version, we must handle three different types of references. These are code references, data references, and external references. The algorithm is described in Algorithm 2. Each link from the old binary to the old versions of the patched code must be removed, and updated to point to the new versions where appropriate. In the event that the new binary relies on an updated version of a library the user must have access to this version. However as long as the new binary has the correct external references, then the quilting procedure still supports it. This becomes particularly useful in the context of dealing with legacy binaries since patching a legacy binary may rely on pieces of new library versions. This means the BinQuilt technique could support multiple versions of the same library as long as BPD provides the correct metadata.

3.2.1 Code section quilting. One important change in newer binaries is that they are built with position independent code (PIC) by default. PIC code became the default on Ubuntu 17.0 across all architectures in 2017 [19]. For security reasons PIC code is designed with relative references so can be deployed anywhere in the address space, but this also means it can be moved from one binary to another. Older code with absolute references on the other hand require finding the corresponding reference in the new binary. *In order to do this the symbols become a point of reference between the binaries.* Each absolute reference can be represented as a symbol and offset. This symbol and offset remains constant across versions even though the absolute addresses will change across versions.

First the links from the original binary to the code that has since been updated need to be removed. The EIR is scanned for any references that point to dead code, and the links are deleted. In most cases these are function pointers. The metadata contains the translation data to update the pointers from the old symbols to the appropriate symbols in the new binary. Next the patched code from the new binary needs to point to the correct places in the original binary. In the event that the patched code depends on code only in the new version these new versions of the code need to be quilted in as well. This allows the quilted binary to be a composite of multiple versions. BinQuilt can quilt all combinations of PIC and non PIC code. The binary patch decomposition metadata contains both the locations of references in the new version that need to be linked, as well as where those links should point. We depend on Egalito to generate the corresponding ELF file.

3.2.2 Data section quilting. The ELF specification splits the binary into data and code sections for security reasons. This introduces some complications to quilting. The code references pointers in the global offset table (GOT) which point to the correct part of the data segment. The second difficulty comes from having to reformat the data segment if it needs to be resized. Since this adjusts the position of every data piece, all entries in the GOT need to be updated.

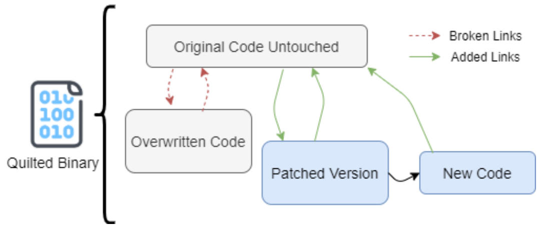


Figure 5: Quilted Binary Construction

Strings are a special case. There are no entries in the symbol table for strings and they are not in the data section with global variables and other data entries. Instead for security reasons they are in a read-only data section and the GOT contains the appropriate offset entry without any associated entry in the symbol table. Instead, BinQuilt iterates through all strings in the old binary for the referenced string in the new binary.

3.2.3 PLT and external reference quilting. In modern executables most libraries are dynamically linked, so instead of direct address entries, the ELF spec uses the procedure linkage table (PLT) to implement lazy loading. Only when the procedure is called does the dynamic linker locate the external symbol in question. In the event the new binary depends on an external symbol that isn't in the original binary, as is the case when the updated functions depend on updated library versions, the PLT needs to be updated with the appropriate entry and the BPD metadata must include the appropriate information for the linking procedure.

4 CASE STUDIES

We use a series of case studies to answer three main questions.

(1) Does Peanut successfully resolve references across versions in real software?
(2) How much new attack space does quilting introduce?
(3) After the patched binary has been created can it be used to verify that both bugs have been patched and no side effects have been introduced?

We evaluated our Peanut prototype on a Dell OptiPlex 7040 with Intel core i7-6700 CPU at 3.4GHz with 32GB memory, running Ubuntu 18.04 64bit, using gcc/g++ version 7.4.0 and python 3.4.7. Peanut is built using CMake version 3.10.2 and Make version 4.1.

4.1 Resolving References

We used Peanut's BPD and BinQuiltto construct quilted binaries for across a variety of projects with a variety of different updates. Table 1 shows the bugs that were patched, a summary of the reason for the code change, the source code size of the change, the number of code section and data section resolutions Peanut made during the quilting procedure.

To test our quilting procedure with the updates from Table 1, we quilted the minimally changed patched version of the binary and the latest available buggy version. We also developed inputs that covered the patched code such that we could test that all references were resolved correctly. If there were any references

Update	Patching Effort	LOC Changed	Code Resolutions	Data Resolutions
curl [5]	Changes how a string is parsed	16+, 16-	31	4
curl [6]	Changes functions arguments and call.	9+, 9-	318	69
libpng [23]	Calculation modification for divide by 0 error	6+, 3-	6	1
wc [9]	Added new function and changed condition check	23+, 2-	298	109
yes [10]	Substantial changes in option parsing	40+, 141-	399	234
ls [3]	Added condition for change in option parsing	1+, 2-	387	380
mv [2]	Adding a conditional check before operation	6+, 0-	204	89
df [7]	Replacing open calls with stat calls	12+,8-	348	164
bs [1]	Changing a loop condition	2+, 1-	296	140
wget [4]	Adding conditional check for log	1-, 2+	10	3
redis [8]	Adding conditional check	1+, 1-	16	8

Table 1: Patch-Testing Dataset

that Peanut didn't resolve correctly during the program run, the program would break. We found that the outputs of the quilted versions were consistent with the outputs of the updated versions. It should be noted that the number of resolutions Peanut requires is independent of the size of the patch but instead dependent upon how central the modified code is to the program's control flow.

4.2 Mathematical Errors: Libpng

In some applications mathematical errors have security implications causing pointer errors or integer overflows. In this instance, an attacker could craft a malicious PNG image that triggers a bad calculation of *row_factor* in Figure 6 [23]. This causes a divide-by-zero error and Denial-of-Service (DoS). After the developer writes the patch and builds the new binary, Peanut will automatically generate the patch metadata for the quilting procedure. Then the user can use Peanut to build a quilted binary and update any software running an old version of libpng. When the quilted binary was tested with a maliciously crafted image, where the *row_factor* was no longer 0, the quilted binary correctly handled the malicious image the same way as the updated version.

4.3 String Parsing: libcurl

String parsing is tricky since there are many corner cases. Figure 7 [5] adjusts Curl's treatment of URLs that end in a single colon.

```
png_check_chunk_length(...) {
...
 size_t row_factor =
-         (png_ptr->width * png_ptr->channels
-          * (png_ptr->bit_depth > 8? 2: 1)
-          + 1 + (png_ptr->interlaced? 6: 0));
+         (size_t)png_ptr->width
+         * (size_t)png_ptr->channels
+         * (png_ptr->bit_depth > 8? 2: 1)
+         + 1
+         + (png_ptr->interlaced? 6: 0);
```

Figure 6: libpng Mathematical Error

```
    ...
+ if(!portptr[1]) {
+   *portptr = '\0';
+   return CURLUE_OK;
+ }
-    if(rest != &portptr[1]) { ...
-    ...
+ *portptr++ = '\0'; /* cut off the name there */
+ *rest = 0;
+ msnprintf(portbuf, sizeof(portbuf), "%ld", port);
+ u->portnum = port;
...
```

Figure 7: Curl URL Parsing

```
+/* Return non zero if a non breaking space.  */
+ static int iswnbspace (wint_t wc) {
+  return ! posixly_correct && (wc == 0x00A0 ...
+ static int isnbspace (int c) {
+   return iswnbspace (btowc (c));
+}
+
wc (args) {
-  if (iswspace (wide_char))
+  if (iswspace (wide_char) || iswnbspace(wide_char))
        goto mb_word_separator;
        ...
-    if (isspace (to_uchar (p[-1])))
+    if (isspace (to_uchar (p[-1]))
+        || isnbspace (to_uchar (p[-1])))
+            goto word_separator;
}
...
```

Figure 8: wc New Function and Refactoring

In the buggy version, Curl incorrectly throws an error and never initiates a valid http request. Figure 7 shows the patch, Peanut resolves all the references in the binary, and sends a valid request.

We test the quilted binary using a specially crafted input and the execution recreates the context that triggered the bug, and then jumps to the patched code upon entering the modified function.

4.4 New Function Refactoring: wc

Peanut even supports changes that introduce substantial refactoring across the entire code base. This includes adding new functions. The new function is treated as new code, and the functions which call the newly implemented function are replaced as well. The quilted binary is tested by providing input from a specially crafted file. The quilted binary acts the same as the updated version proving it successfully integrates the patch into the old binary.

4.5 Increased Attack Surface

By quilting the binary with new code, we introduce a new attack vector that an attacker could potentially exploit. As such it's important in the quilting procedure to minimize the attack surface. However since we only add the bare minimum code necessary to create the quilted binary even if the update involved including new versions of libraries our code only quilts the updated functions and its dependencies. Table 2 shows the minute increased attack surface. The maximum was measured at 14% and the minimum at .003%. Egalito keeps the additional attack surface to a minimum. Since Egalito raises the binary to an intermediate representation

Update	Original Size (KB)	Quilt Increase (KB)	Pctg Increase (%)
curl [5]	663.4	.687	.104
curl [6]	663.5	9.6675	14.569
libpng [23]	942.2	.237	.025
wc [9]	222.0	5.941	2.675
yes [10]	148.5	10.669	7.184
ls [3]	623.5	7.497	1.202
mv [2]	613.5	4.303	.701
df [7]	423.3	5.971	1.41
bs [1]	295.1	5.648	1.914
wget [4]	8740	.303	.003
redis [8]	1086	.138	.013

Table 2: Quilting Overhead

Number of Requests	Number of Clients	Original (s)	Quilted (s)	Updated (s)
100K	50	5.19	5.14	5.12
200K	100	10.43	10.44	10.42

Table 3: Quilted Performance

and then regenerates the binary, there are no additional jump instructions, trampolines, or extra pieces from the patched version of the binary or its accompanying libraries. By using binary regeneration instead of traditional binary rewriting mechanisms, even large transformations introduce minimal overhead.

4.6 Side Effect Verification

The main advantage of quilting is that it prevents updates from unintentionally introducing unwanted side effects. Developers can test for most functionality related side effects, but side effects specific to the user environment like performance and scalability are difficult to diagnose with deploying the new version or at least a mock deployment. Developers can't test for these types of problems.

To simulate this we conducted a case study using the redis webserver. A version of redis [8] had a bug in which connecting via a monitor thread, and sending a specific message caused the server to crash. This is a critical level vulnerability that would need to be patched immediately.

To simulate the patch first we quilted the binary in question and ran the same patch quilting procedure. We then tested the patch's functionality with the predesigned test scenario to make sure that it successfully integrated the patch the same way we did in the other studies. We then ran Redis' included benchmark commonly used to test Redis' performance on the original, quilted, and updated binary. We found no evidence that quilting introduces any performance overhead as shown in Table 3.

5 LIMITATIONS

Peanut depends on good practice in developer commits. If they include extraneous code not related to the designated bug-fix in the same commit, Peanut would automatically include this extra information in the diffs. In other words, in order for Peanut to be useful, developers need to make modular commits that distinguish bug-fix commits from other commits containing enhancements and other changes. Peanut also relies on having access to the symbols in the build process. Of course these are stripped for deployment but should be available in the developer environment.

Peanut has some approach limitations so it cannot accommodate arbitrary code changes. The main approach limitation is in dealing with data structure modifications. Peanut makes no attempt to track data types, so quilting in a new data type with the current implementation would be impossible.

The current Peanut implementation does not support changes to macros. Macros are inlined to the code sections, so this information would not be available in the binary. Attempting to find the location of macros in a binary is impossible since this information is lost in the developer's build process. Therefore Egalito can't apply the appropriate transformations. Similarly in code that is highly optimized, as would be expected in production level code, this could cause a problem with inlined functions. Egalito would have no way of finding the locations of the inlined functions.

We asked an independent team of three students to construct a set of known C/Linux bugs that had already been fixed, 2016–2019, and to write appropriate test cases. Although we encouraged them to include some security vulnerabilities, we placed no restrictions on what bugs they could look for, so the collected bugs may not be representative of bugs found in the wild that users would deem critical to patch.

6 CONCLUSION AND FUTURE WORK

Through this work we showed that we are able to use binary patch decomposition (BPD) in conjunction with binary quilting (BinQuilt) to combine updated and older versions of a binary to create a variant of the older binary that only includes the desired changes from the updated version. Initial investigations with our Peanut prototype indicate that we can apply this approach to real binaries based on real updates and real build processes. We ran case studies across multiple programs with a variety of update types and sizes. Our case studies suggest that Peanut could accommodate production updates without introducing side-effects, relative to the original binary, beyond the designated patch. Further, Peanut's leveraging of Egalito's binary regeneration avoids the performance overhead, relative to the original binary, common in other approaches to binary rewriting.

In the future we plan to conduct a more thorough evaluation of our techniques, with more types of updates, in which we compare to both the overhead introduced by other binary rewriting tools. We have only tested with the minimal patch size for making an update of limited scope, but there's no reason our technique couldn't accommodate composite version updates of a significantly larger scope, i.e., multiple independent patches and/or a series of bug-fixes that work together. Given access to the proper dataset we would like to test this with legacy binaries that are especially relevant and hard to deal with.

ACKNOWLEDGMENTS

The Programming Systems Laboratory is supported in part by NSF CNS-1563555, CCF-1815494 and CNS-1842456.

REFERENCES

[1] 2017. Running b2sum with −check option, and simply providing a string "BLAKE2" . https://debbugs.gnu.org/cgi/bugreport.cgi?bug=28860.

[2] 2018. 'cp -n -u' and 'mv -n -u' now consistently ignore the -u option . https://github.com/coreutils/coreutils/commit/\protect\discretionary{\char\hyphenchar\font}{}{}7e244891b0c41bbf9f5b5917d1a71c183a8367ac.

[3] 2018. ls -aA shows . and .. in an empty directory; . https://debbugs.gnu.org/cgi/bugreport.cgi?bug=30963.

[4] 2018. Simple fix stops creating the log when using -O and -q in the background . https://github.com/mirror/wget/commit/\protect\discretionary{\char\hyphenchar\font}{}{}7ddcebd61e170fb03d361f82bf8f5550ee62a1ae.

[5] 2019. Curl String Parsing Bug. https://github.com/curl/curl/pull/3365.

[6] 2019. Curl String Parsing Bug. https://github.com/curl/curl/pull/3381.

[7] 2019. df coreutils library function. https://github.com/coreutils/coreutils/commit/b04ce61958c.

[8] 2019. Redis Monitor Request Causes Crash. https://github.com/antirez/redis/commit/e2c1f80b.

[9] 2019. wc Special Character Bug. https://github.com/coreutils/coreutils/commit/a5202bd58531923e.

[10] 2019. yes coreutils library function. https://github.com/coreutils/coreutils/commit/44af84263e.

[11] Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: Accurate Comparison of Binary Executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop* (Rome, Italy) *(PPREW '13)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/2430553.2430557

[12] Agner Fog. 2019. *Calling conventions for different C++ compilers and operating systems*. Technical University of Denmark. https://www.agner.org/optimize/calling_conventions.pdf

[13] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Information and Communications Security*, Liqun Chen, Mark D. Ryan, and Guilin Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 238–255.

[14] Github. [n.d.]. GitHub. https://github.com/. (Accessed on 07/14/2020).

[15] William H Hawkins, Jason D Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W Davidson. 2017. Zipr: Efficient static binary rewriting for security. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 559–566.

[16] J Hiser, A Nguyen-Tuong, W Hawkins, and M McGill. [n.d.]. M. Co, and J. Davidson. Zipr++: Exceptional Binary Rewriting. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*. 9–15.

[17] He Huang, Amr M. Youssef, and Mourad Debbabi. 2017. BinSequence: Fast, Accurate and Scalable Binary Code Reuse Detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 155–166. https://doi.org/10.1145/3052973.3052974

[18] H. Jeong, J. Baik, and K. Kang. 2017. Functional level hot-patching platform for executable and linkable format binaries. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 489–494.

[19] Steve Langasek. [n.d.]. Ubuntu Foundations Team - Weekly Newsletter. https://lists.ubuntu.com/archives/ubuntu-devel/2017-June/039816.html

[20] Jiang Ming, Meng Pan, and Debin Gao. 2013. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *Information Security and Cryptology – ICISC 2012*, Taekyoung Kwon, Mun-Kyu Lee, and Daesung Kwon (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 92–109.

[21] Alfred Ng. 2018. How the Equifax hack happened, and what still needs to be done. https://www.cnet.com/news/equifaxs-hack-one-year-later-a-look-back-at-how-it-happened-and-whats-changed/

[22] A. Ramaswamy, S. Bratus, S. W. Smith, and M. E. Locasto. 2010. Katana: A Hot Patching Framework for ELF Executables. In *2010 International Conference on Availability, Reliability and Security*. 507–512.

[23] Red Hat Bugzilla – Bug 1599943. 2019. libpng: Integer overflow and resultant divide-by-zero. https://bugzilla.redhat.com/show_bug.cgi?id=1599943. CVE: https://nvd.nist.gov/vuln/detail/CVE-2018-13785.

[24] Wang, Shi-Chao, Liu, Chu-Lei, Li, Yao, and Xu, Wei-Yang. 2017. SemDiff: Finding Semtic Differences in Binary Programs based on Angr. *ITM Web Conf.* 12 (2017), 03029. https://doi.org/10.1051/itmconf/20171203029

[25] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 133–147. https://doi.org/10.1145/3373376.3378470

[26] zynamics. [n.d.]. BinDiff. https://www.zynamics.com/bindiff.html. "Accessed: 7/9/20".