SOLDER: Retrofitting Legacy Code with Cross-Language Patches

Ryan Williams
Northeastern University
Boston, USA
williams.ry@northeastern.edu

Anthony Gavazzi
Northeastern University
Boston, USA
gavazzi.a@northeastern.edu

Engin Kirda
Northeastern University
Boston, USA
e.kirda@northeastern.edu

***Abstract*—Internet-of-things devices are widely deployed, and suffer from easy-to-exploit security issues. Due to code and platform reuse, the same vulnerability oftentimes ends up affecting a large installed base. Because patch deployments tend to be focused on server-side vulnerabilities, client software in large codebases such as Apache may remain largely unpatched, and hence, vulnerable. This problem is exacerbated by the prevalent use of some of these large codebases in IoT deployments, making their use more widespread.**

In this paper, we address this issue of leaving latent vulnerabilities in legacy codebases. We propose SOLDER, a framework to patch or retrofit legacy C/C++ code by replacing any target function with a newly-implemented one in a safe language such as Rust. This allows application users to freely patch their software in a safe language whenever a patch becomes available, without the need for waiting on developers to release an official patch. When dealing with mission-critical systems, it may be infeasible to wait the months it takes, on average, for patches to be released. Internally, SOLDER performs this function swapping on LLVM bitcode, and can either target source code directly to generate the IR, or if source is unavailable, can lift a binary to LLVM bitcode before running the analyses and then recompiling back to binary.

Evaluation on 5 popular codebases shows that SOLDER produces a valid patched binary that is 2.3% smaller, on average, and mitigates 11 known exploitable vulnerabilities, while introducing limited overhead and no new bugs.

I. INTRODUCTION

While there have been efforts to expedite the process for creating and implementing software patches in response to vulnerability disclosures [1], the average time-to-patch for critical vulnerabilities has actually increased up to 205 days as of May 2021 [2]. These vulnerabilities also have a long average life expectancy of 6.9 years, where they can remain latent on a mission-critical system [3], [4]. However, once a vulnerability has been found, the median time to develop a fully-functional exploit is much faster, at 22 days [4]. To date, most patch deployments have been focused on server-side vulnerabilities [5]–[7], leaving client-side applications such as browsers, firmware, or SSH clients more open to threats.

A problem with updating or patching client-side legacy codebases is that the analyst may be unfamiliar with the original implementation language, or the source code and compilation environment are not available. Moreover, the choice of implementation language can greatly impact the quality of the patch, as software written in C/C++ tends to suffer from some every-day programming issues, including dangling pointers, memory leaks, data races, and buffer overruns/underruns [8]. Certain classes of vulnerabilities are also associated with these languages, such as memory safety issues. A survey done by Microsoft shows that since 2006, around 70% of the vulnerabilities addressed through a security update each year continue to have memory safety issues [9].

A language such as Rust can, by design, help mitigate these classes of vulnerabilities. As a memory-safe language, Rust has been considered by many companies to patch their software by creating Rust libraries for new software versions, developing APIs to invoke Rust code, and replacing anywhere from parts of their code to entire codebases with Rust. For example, Google and Microsoft have started to investigate the use of Rust for parts of their codebases and began development with a mix of C#, Rust and constrained C++ [10]. Mozilla has integrated the Quantum CSS and WebRender rendering engines, both of which are written in Rust, into Firefox [11]. Additionally, Facebook has started to build an unofficial Rust support team since 2017 to investigate the use of Rust at Facebook [12].

However, rewriting an entire software component with a new implementation language is resource-consuming work. Because patching security-critical software is time-sensitive, and rewriting components of legacy code requires substantial manual effort, we provide a framework that enables application users to update legacy code with Rust-based patches without needing to involve the application developers. SOLDER provides a platform for application users to target given functions in a codebase, and replace their definitions with user-provided patches implemented in a language of their choosing. Based on a study by Li et al. [13], the majority of security patches are localized and limited in scope, unlike general software patches. It is for this reason that we chose to implement patching at the function-level.

In this paper, we introduce SOLDER, a novel framework for patching and retrofitting legacy code with functions implemented in *safer* languages, such as Rust, without the need for waiting on developers to issue a patch. Using this method, technically-sophisticated application users and mission-critical organizations can incrementally patch their client software as soon as a patch is available even without access to the application source code. Using SOLDER, we are able to incrementally

patch legacy codebases at a functional granularity with any language that has an LLVM front-end.

In summary, our work makes the following contributions:

1) We present a new foundation for cross-language program patching and diversification that allows users to effectively retrofit legacy code;
2) We provide a platform that can also be used for incrementally updating legacy code to a more modern language;
3) We implement a light-weight symbolic execution mechanism for verifying the validity of user-defined patches;
4) We evaluate SOLDER on five real-world programs that cover a wide range of usage scenarios including IoT and embedded devices and demonstrate its efficacy;
5) We include a security analysis of SOLDER which shows patches implemented in a language different from that of the project can still mitigate 11 real-world CVEs while reducing the number of ROP gadgets that are present.

II. BACKGROUND

A. Motivation of Our Work

SOLDER's goal is to provide a foundation to patch legacy code and promote software diversity among homogeneous software deployments. A problem with updating or patching legacy codebases is that the analyst may be unfamiliar with the original implementation language, or that there may be features needed from another, perhaps newer, language [14]. In other cases, the analyst may need to apply a patch immediately for a critical issue as disclosed in a Common Vulnerabilities and Exposures (CVE) disclosure, where a patch is not yet provided. Waiting on an organization to release a new, patched application may be infeasible, but using SOLDER, the analyst can target the function in the application they wish to replace with an updated implementation. SOLDER operates entirely on LLVM bitcode as LLVM is a set of compiler technologies that is designed around a language-independent intermediate representation (IR), which allows us to develop a variety of bespoke transformation and analysis passes.

Using SOLDER, one is able to *incrementally* update legacy codebases at a functional granularity. That is, one can take a legacy C codebase and update individual functions within it in languages such as C++ or Rust. Because SOLDER is implemented at the LLVM IR-level, it is language-agnostic as long as the target codebase has an LLVM front-end (e.g., clang, rustc, gollvm). While this means that we primarily target C family languages in our prototype, we can still extend it to other languages by implementing an LLVM front end for them. SOLDER applies patches at the function granularity due to the tendency of security patches to be very localized. For example, the vulnerability CVE-2019-11779 for Mosquitto shown in Figure 1 can be completely patched by updating the `mosquitto_pub_topic_check()` function.

Because many manufacturers now opt to stop providing system update services for their obsolete models, millions of vulnerable, unpatched devices remain in use [15]. Being able to patch legacy or obsolete software is essential to mitigate known vulnerabilities, and is especially challenging with fragmentized devices. When using SOLDER to update a target project's language, the user also acquires the added benefit of inherently minimizing the attack surface (see Section V-D). This is worth noting as a larger attack surface may make it easier to mount attacks such as ROP and code reuse [16].

```
49    int mosquitto_pub_topic_check(const char *str)
50    {
51            int len = 0;
52  + #ifdef WITH_BROKER
53  +         int hier_count = 0;
54  + #endif
55            while(str && str[0]){
56                    if(str[0] == '+' || str[0] == '#'){
57                            return MOSQ_ERR_INVAL;
58                    }
59  + #ifdef WITH_BROKER
60  +                 else if(str[0] == '/'){
61  +                         hier_count++;
62  +                 }
63  + #endif
64                    len++;
65                    str = &str[1];
66            }
67            if(len > 65535) return MOSQ_ERR_INVAL;
68  + #ifdef WITH_BROKER
69  +         if(hier_count > TOPIC_HIERARCHY_LIMIT) return MOSQ_ERR_INVAL;
70  + #endif
71
72            return MOSQ_ERR_SUCCESS;
73    }
```

Fig. 1. Snippet of a vulnerable function for CVE-2019-11779 in Mosquitto MQTT. If `str` contains more than 65535 '/', a stack overflow will occur. The patch modifications are prefixed with '+'.

We do not claim that SOLDER guards against every possible vulnerability, nor that it cannot be bypassed in individual cases by a skilled, motivated attacker. Its goal is to allow for users to patch application codebases or binaries without involving the developers. This provides a means to patching and retrofitting legacy code. While SOLDER can effectively mitigate some known CVEs, its utility is in its general ability to incrementally update legacy code with safer, more modern languages.

B. Retrofitting Legacy Code

Commercial off-the-shelf (COTS) applications and legacy codebases are both primary targets for attacks as they tend not to provide sufficient security features. To mitigate this attack vector, there have been efforts in retrofitting COTS binaries and legacy code, typically with *wrappers* [17]. These generic wrappers are meant to act as an intermediate layer that can observe and modify data passing through the interfaces, acting as a form of sanitization and validation [18]–[20]. The main challenge with retrofitting some legacy code is identifying where these sensitive operations occur [21].

As retrofitting legacy code is typically done manually, program analysis has been shown to help this process by guiding analysts to *critical* program points [22]. However, this still necessitates making modifications directly to the target codebase, and having a strong understanding of the system,

whereas SOLDER allows developers to provide a functional patch without needing to understand the holistic codebase.

C. Binary Patching

Binary patching provides a means to fixing bugs in applications when part of an application's source code has been lost, or when an environment the application requires is not available. For example, many embedded systems and bare-metal systems do not have the available source code, or the original compilation toolchain, which makes patching them difficult, or even impossible. Although tools such as Ghidra or IDA Pro provide a robust platform for reverse engineering and binary modification, the requirement of having an advanced understanding of assembly language limits its general applicability, and the manual work required is non-trivial. For example, Microsoft has spent a significant amount of effort to manually fix CVE-2017-11882 using tools such as these [23].

SOLDER provides a novel way to perform program patching without directly touching the assembly language. We also assume that we do not have access to paid tools such as IDA Pro for finding patch insertion points like in BinPatch [24]. Instead, we utilize the expressiveness of LLVM bitcode, and perform all of SOLDER's operations at that level. While other tools like BinPatch modify binaries by inserting long jumps to the patch code, we focus on actually removing the vulnerable code. This addresses the potential security issue BinPatch introduces with long jumps violating security requirements like control flow integrity (CFI). This makes binary patching more feasible, and saves both time and human effort.

III. DESIGN OVERVIEW

A. Objectives and Assumptions

SOLDER sets out to provide a semi-automated framework for swapping out discrete components within a codebase using user-provided, or publicly-available source patches. Our tool currently works at the function-level by targeting specific functions within a project, removing all references to that function, and replacing them with the new target functions. While there are various use cases for SOLDER, such as program hardening, program diversification, program understanding, or incrementally updating a codebase's implementation language; we are primarily working to address the gap between vulnerability disclosure and patch availability. Using SOLDER, a user can implement their own patch for a known-vulnerable function as soon as they have one working, or they can use a publicly available patch that is yet to be pushed into an automatic update.

Currently, we assume that source code is available for the codebases that we target. However, SOLDER can also work on binaries, in which case there are some limitations (see Section VI-C). SOLDER does not, however, modify the source code of the project itself. Instead, it makes a set of LLVM passes that resolve the functions to swap at compilation and linking time. While our use case here assumes access to source, we have done a number of tests to show that SOLDER is capable of applying this patching technique to binary targets without access to the source. We also assume that any target codebases can be compiled with an LLVM frontend (e.g., Clang), as our analyses operate on bitcode. We also assume that any patches from the NVD database are correct, but user-provided patches need to be verified.

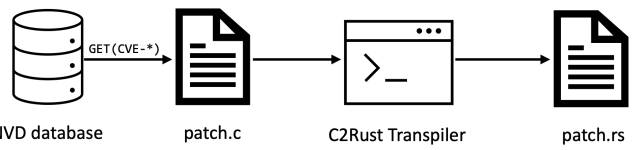

Fig. 2. Workflow of SOLDER's patch generation component for translating publicly-available native patches to a Rust equivalent.

An overview of SOLDER's workflow is shown in Figure 3. The following sections go into more detail on each of the corresponding steps shown there.

B. Patch Generation

The first thing SOLDER requires is a patch for a given project. Currently, we rely on patches detailed in the National Vulnerability Database (NVD). If there is a source patch provided via a git commit, we take the diff to find the parent function, and translate the whole function into Rust using C2Rust [25]. The Rust patch is put into a separate file and compiled to LLVM bitcode while the target project is being compiled or lifted to the same IR. While this technique relies on the accuracy of a provided patch, this is the simplest approach to using SOLDER in an automated way. Our target use case, however, would typically be that the user has developed an out-of-source patch independently of the application developers, allowing them to implement the patch without waiting for one to be rolled out in an update.

While it is true that using a direct translation to Rust would typically use primarily *unsafe* code, we found that despite this, a direct translation using unsafe Rust was still better than keeping the native legacy language. Implementing updated code in Rust provided a reduction in attack surface (via ROP gadgets) even when leaving the vulnerability latent.

If a patch is available from NVD or another public source, we simply transpile that patch to Rust, and inject that into the project with SOLDER. Otherwise, in the case where a vulnerability is disclosed, but yet-to-be patched, we can either: (*i*) write a patch on our own in the language of our choice; or (*ii*) transpile the vulnerable function to a Rust equivalent and use that as a patch. In the second case, we may not be mitigating the vulnerability, but we are altering the attack surface, while potentially implicitly patching the vulnerability if it is something memory-related, for instance, as Rust provides strong memory safety guarantees. There is a potential trade-off here where applying an unvetted patch prematurely may result in leaving a vulnerability present or even introduce a new one. However, we found that refactoring legacy code to Rust always had a net benefit. The workflow of the patch generation step is shown in Figure 2.

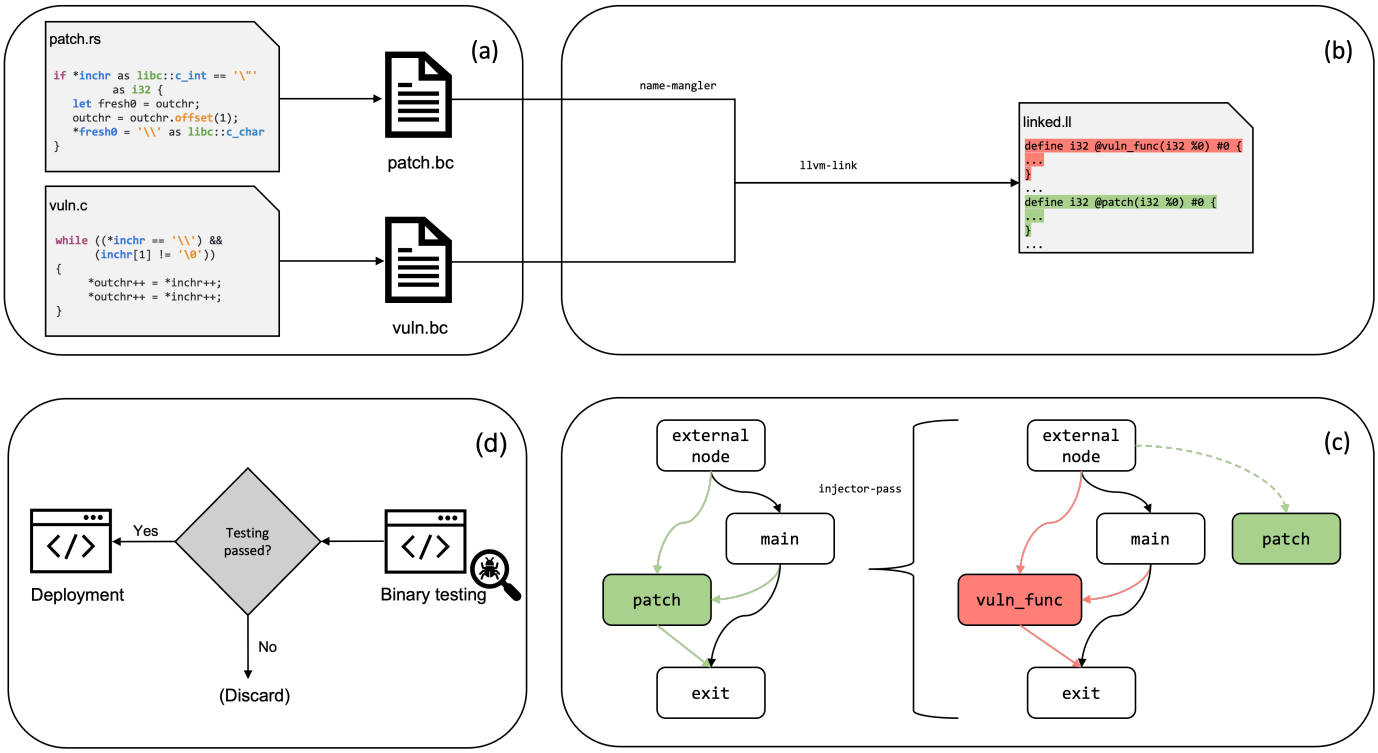

Fig. 3. Overview of SOLDER's workflow shown in its 4 stages. The vulnerable function `ap_escape_quotes()` from Apache (CVE-2021-39275) and a patch written in Rust are first compiled to bitcode (**a**). Next, they are linked with the symbol names in the patch mangled (**b**). The patch injection step swaps out the vulnerable function with the patch (**c**). The compiled binary is then tested before being ready for deployment (**d**).

1) Patch Testing: Once we have a candidate patch, we test the patch to provide some guarantees that it will not be introducing any previously-unknown bugs, or semantic errors. First, we run any unit tests which may be available with the target program. Next, using the LLVM-based symbolic execution engine, KLEE [26], we run symbolic execution against the patch that we generated. Because symbolic execution scales poorly, we only run it against the patch in isolation instead of in the context of the entire program. Since we chose to use Rust-based patches, in order to test them using KLEE, we require LLVM bitcode of the core Rust libraries. This testing gives us a quick result indicating the presence or absence of any bugs, as KLEE attempts to explore every path in the program.

We initially explored using QSYM [27] as well as KLEE. Using a greybox fuzzer approach had some drawbacks such as needing seed input, and issues working on network protocols. We opted for using symbolic execution over fuzzing here because we are operating on the LLVM IR level, which made KLEE a good option in our workflow. We are also testing patches that are not too large, so the typical path explosion problem was not encountered in our tests. The other added benefit is that with KLEE, whenever a new path is explored, we can output a new test case, and these test cases can then be replayed against the final binary that we produce.

For our testing setup, SOLDER only runs KLEE against a target patch for 30 seconds before terminating. We chose this short amount of time as it does not incur too much overhead, while also providing 84% instruction coverage on average, as shown in Table VI.

C. Compilation to Bitcode

Once we have the patch from our previous step, it is time to compile that and the target project to LLVM bitcode. SOLDER operates on bitcode as opposed to source due to the fact that it allows multiple source languages to be mutually-intelligible. For instance, swapping out a function in a C-based project with a new function written in Rust is straightforward when that operation happens at the bitcode level. For patches written in C/C++ or Rust, this step is straightforward, as it only requires that we provide certain compile-time flags to rustc or Clang. However, when compiling the target program to bitcode, we must accommodate both the case where we have access to source/build files and the case where we only have the compiled binary.

When we have access to the project's source, it is just a matter of wrapping the build process to force emitting LLVM bitcode. This is done by setting the compilation flags for Clang to output the intermediate files. In the case of some larger projects, we also used *Whole Program LLVM* (WLLVM) [28] for bitcode extraction, and eventually moved to integrating this into SOLDER as it simplified the overall process. Using WLLVM simplified the building to LLVM bitcode process because it acts as a drop-in replacement for `gcc` in any build system. This saves a lot of time in the case of projects where

we would otherwise need to manually modify the build system to output bitcode.

In the case of no access to source, we use the tool RetDec [29] for lifting the binary to LLVM bitcode. As this lifter supports most executable file formats and architectures, it provides ample support for SOLDER to patch binaries without access to source code. While we tested using SOLDER without access to source, those tests were constrained to smaller programs, and evaluation on more real-world programs is left to future work. SOLDER's compilation to bitcode step is shown in Figure 3(a).

D. Component Linking

To target functions for swapping in from another file, we use the LLVM built-in, `llvm-link`, to link together each of the components we compiled to LLVM bitcode. To avoid errors for multiply-defined symbols, we run another pass at this stage that mangles function names. This obviates the need for worrying about reusing function names in the user-provided patch. Details on this pass are outlined in the next subsection. However, the intermediate *linked* file that is created here will be larger as it may include multiple definitions of the same functionality as they were necessary for compiling the patch independently. These superfluous functions will be properly cleaned up in the subsequent steps. Once everything is linked into one bitcode file, SOLDER is then ready to invoke the LLVM pass that swaps the given function(s) with the functions in the provided patch. The component linking process is shown in Figure 3(b).

1) Function Name Mangling: Because we may be trying to link arbitrarily many files with matching symbol names (think multiply defined `main` function symbols), we first have to run an LLVM pass that prepends function names and global variables with their component file ID, or some other unique ID specified by the user. Once we mangle the names accordingly, we are able to successfully link all the bitcode files. This name mangling pass also accepts an `exclude` parameter that allows the user to provide a list of functions not to mangle. This is useful for specifying which source file has the `main` function that we wish to keep so we do not have to specify a program entrypoint later. This pass also accepts the `prefix` parameter that allows the user to specify a static value to prepend to function and global variable names for mangling. For example, a function named `read_config` in the patch file would be transformed to `_ignore_read_config` when the prefix is set to `ignore`.

E. Patch Injection

The LLVM pass that handles the actual swapping of functions takes the parameters `target` and `replacement`. This is used to target the function we wish to swap and point to the implementation of its candidate replacement. The only requirement here is that the function parameters and return type match. We cannot, for instance, replace an `int add(...){...}` function with a `float add(...){...}` function as they accept different types when they are called. We can, however, replace a C-defined `int add(...){...}` with a Rust-defined `pub fn add(...) → u32 {...}`. As long as the parameters and return types match, the rest of the function's implementation is up to the discretion of the patch author.

Once the function patching has taken place, we then remove any extraneous functions that were linked in using SOLDER's *def-use* analysis pass. The Rust patches tend to need more helper functions defined, but some of these are never used, and can make our binaries larger than they need to be. For instance, writing a standalone patch for an addition function in Rust results in an extra four functions when outputting LLVM bitcode. The first of these functions is `rust_eh_personality`, which is a language item used by the failure mechanism of the Rust compiler. While this is necessary in our initial patch compilation step, we can safely remove it at this stage as all it does is add bloat to our output binary. By using our data-flow analysis to find dependencies between functions, we can simply remove those linked functions that have no references without breaking any other functionality. The workflow of patch injection is shown in Figure 3(c).

F. Build Testing

Once all the preceding steps have taken place, we can now generate the object code from the patched bitcode. This step is built on LLVM's static compiler, `llc`. SOLDER outputs the bitcode to an object (`.o`) file, then reuses the compile commands from the original project. The compile commands can be extracted either by using the flag `-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` for `CMake`, or by wrapping the compilation process using Bear [30]. In the case that we do not have access to the source and build environment, this last step is scripted using Clang. SOLDER attempts to compile the object file without any options, then finds any missing dependencies and attempts to resolve those incrementally using a trial and error approach.

Once we have recompiled the binary, we run any test suites that are provided with the codebase in the case of source and tests being available. This process is shown in Figure 3(d).

IV. IMPLEMENTATION

Our SOLDER prototype implementation is built as a set of compilation passes on LLVM [31]. SOLDER works on C/C++ and Rust programs that are compiled into LLVM bitcode.

For our patch generation stage, we extended the tool `cve-search` [32] to follow referenced links in order to automate getting source-level patches. For translating the patches from C to Rust, we use C2Rust [25]. Next, for the patch testing, we use KLEE [26], which operates on LLVM bitcode, just like the rest of SOLDER. KLEE did not require any modifications to work in our system.

The linking stage of SOLDER requires handling multiply-defined symbols. To facilitate this step, we implemented the `name-mangler` LLVM pass. Next, SOLDER targets a given

TABLE I
CHARACTERIZATION OF CODEBASES. WHERE #CVES IS THE NUMBER OF CVES TESTED FOR EACH CODEBASE, AND #FUNCTIONS IS THE TOTAL NUMBER OF FUNCTIONS AVAILABLE TO TARGET.

Program	#CVEs	#LOCs	#Functions	Source Language
OpenSSL [34]	2	165,630	1,773	C
OpenSSH [35]	3	130,421	1,586	C
Apache [36]	2	52,600	2,490	C
Mongoose [37]	3	55,904	275	C
Mosquitto [38]	1	38,360	387	C

TABLE II
INCURRED AVERAGE OVERHEAD OF RUNNING EACH OF SOLDER'S STAGES ON OUR TARGET CODEBASES FIVE TIMES. STANDARD DEVIATION (σ) FOR EACH IS SHOWN IN PARENTHESES.

Program	S_{bc}	S_{link}	S_{swap}	S_{exit}
OpenSSL	3.8s (0.1)	1.3s (0.08)	1.7s (0.2)	199.4s (1.1)
OpenSSH	2.0s (0.1)	0.7s (0.1)	0.8s (0.1)	33.5s (0.9)
Apache	1.9s (0.1)	0.4s (0.08)	0.4s (0.1)	55.6s (0.3)
Mongoose	1.5s (0.2)	0.04s (0.01)	0.05s (0.02)	1.7s (0.1)
Mosquitto	1.7s (0.08)	0.2s (0.08)	0.3s (0.05)	5.1s (0.1)

TABLE III
AVERAGE COMPILATION TIME USING SOLDER VERSUS GCC (IN SECONDS) OVER FIVE RUNS. STANDARD DEVIATION (σ) SHOWN IN PARENTHESES.

Program	Compile-time (SOLDER)	Compile-time (gcc)
OpenSSL	206.2s (0.8)	181.0s (1.0)
OpenSSH	37.0s (1.9)	31.8s (1.3)
Apache	58.3s (1.0)	55.5s (2.1)
Mongoose	3.3s (0.3)	1.6s (0.3)
Mosquitto	7.3s (0.3)	5.1s (0.1)

function and replaces it with a new implementation. This is implemented as another LLVM pass, `injector-pass`.

Lastly, for evaluation purposes, we built on RetDec [29] to test lifting binary to LLVM bitcode and back. For evaluating ROP gadget counts, we used the ROPgadget tool [33].

SOLDER is implemented with 775 lines of C++, 522 lines of Rust, 304 lines of Python, and 126 lines of Bash, not including the tools we used unmodified.

V. EVALUATION

In this section, we evaluate the prototype of SOLDER. We conducted: (*i*) micro-performance benchmarks to measure the overhead of each step of patching; (*ii*) macro-performance tests on **5** real-world programs that run on Linux-based embedded systems to measure SOLDER's overall overhead; (*iii*) tests to show the correctness of the patched binary; and (*iv*) a security analysis of how SOLDER can mitigate known vulnerabilities. The codebases evaluated are characterized in Table I.

A. Micro-Performance Tests

The compile-time overhead of patching target programs can be broken down into four steps as shown in Table II: (*i*) compilation of the target codebase and patch to LLVM bitcode (S_{bc}); (*ii*) linking the patch into the target project (S_{link}); (*iii*) swapping the patched function in place of the original (S_{swap}); and (*iv*) recompiling the patched binary (S_{exit}) at the end of the workflow.

As we can see in Table II, the two stages that incur the most overhead are bitcode generation, and recompilation. Because some projects are larger, the bitcode generation step (S_{bc}) can take considerably longer as we see with Mongoose versus OpenSSL. The main bottleneck here is that we must output bitcode for the Rust patch as well as the target project. While the Rust patch is typically very small, compiling the target project itself to LLVM bitcode may be non-trivial. The last step of recompilation (S_{exit}) can also potentially slow down the process; however, it is always roughly the same overhead as simply compiling the target project (see Table III). It should be noted that the primary component of SOLDER which performs the function swapping always incurs a low overhead compared with the compilation steps.

We omit the step of *patch generation* as in our model, this stage may be manually done by the user. However, there are cases where we have automated the process of generating a patch in Rust from a publicly-available patch in C/C++, as shown in Figure 2. In our evaluation, the patches we used were all pulled from NVD and transpiled from their native language to Rust, following this automated approach. Furthermore, we tested replacing arbitrary functions in each codebase for the purpose of incremental updating, and not vulnerability mitigation. In these cases, SOLDER is able to directly translate a target function into a Rust equivalent with negligible overhead.

B. Macro-Performance Tests

For our 5 evaluation targets, we measured the end-to-end overhead of using SOLDER. While all 5 targets were written in C (see Table I), they could just as easily have been implemented in any other language that has an LLVM frontend. We chose these projects due to their ubiquity, especially in embedded/IoT deployments. Working with these large, C-based projects is also more in-line with our model of patching or retrofitting legacy codebases.

Compile-time Overhead. We measure the whole compilation process of running SOLDER on our 5 target programs. On average, SOLDER incurs a 36.9% overhead. Table III shows the difference in compile times using SOLDER versus the default compiler for each project.

Runtime Overhead. As SOLDER provides patches directly to the target codebase, the only overhead incurred is at compilation time. When analyzing runtime statistics, there is no discernible difference in performance between running the binary with a native patch or a SOLDER-based patch.

Binary Size Measurement. Because the programs we target typically run on resource-constrained devices, it is important that SOLDER does not result in a binary that is too large. Table IV shows the difference in size of the natively-patched binary and the Rust-based patch implemented from SOLDER. The only instance where SOLDER generates a larger binary is with OpenSSH. This was due simply to the size of the patch; however, we believe that if we manually wrote the patch in a more optimized way, the resulting binary would be smaller. Overall, we see that on average, using SOLDER results in a

TABLE IV
SIZE OF SOLDER-PATCHED BINARY VERSUS THE NATIVELY-PATCHED COUNTERPART.

Program	Binary Size (SOLDER)	Binary Size (Native)
OpenSSL	4600kb	4800kb
OpenSSH	2700kb	2600kb
Apache	1100kb	1100kb
Mongoose	80kb	90kb
Mosquitto	600kb	600kb

binary that is **2.3%** smaller than its native counterpart. In our tests, we found that the reason for smaller binaries was due to SOLDER's *def-use* analysis stage. That is, after the Rust-based patch is inserted, the def-use analysis stage then looks for any unused functions that can be safely removed. In these cases, we found that some Rust-based patches do not utilize functions that the original patch did. When this is the case, those functions in question can be safely removed from the codebase, thus reducing the overall compiled size.

C. Correctness of Patched Binary

In order to provide guarantees on the correctness of the patched binaries SOLDER produces, we re-run any unit tests that are provided with the project. Table V shows the number of tests run for each codebase, and the respective number of crashes observed. However, if test suites are not available, or if we are assuming no access to source at all, then we need to rely on symbolic execution. SOLDER uses KLEE to thoroughly test the user-provided patch before swapping it into the target. Table VI shows the amount of test coverage provided by KLEE and the errors reported for each.

For all of the included unit tests we ran against our targets, there were no errors reported for our SOLDER-patched version. However, when running KLEE against one of our individual patches, we found an out-of-bound pointer error in a Mongoose patch. Because our patch was a direct Rust translation of the one provided by the Mongoose developers, we also tested their patch, and found that the same error was triggered. This shows that while SOLDER did not introduce any *new* errors, those that are latent can still be present when the patch is not adequately vetted.

To provide a real-world test on the correctness of a patched binary, we compiled a patched version of OpenSSH, and ran `sshd` on a production server for two weeks without any reported errors, or issues from users. Over the course of this two week-long experiment, we had an average of 27 SSH connections to the server per-day. This test gives us confidence that our version of OpenSSH's `sshd` with Rust-based patches does not introduce any new issues that were previously not present.

D. Security Analysis

When evaluating SOLDER, we targeted specific commits for the codebases where a known CVE was present. A summary of the CVEs we targeted per-codebase is shown in Table VII. The CVEs we targeted were all relatively localized as far as how many functions needed to be modified to provide a working patch. Our evaluation made two observations: *(i)* the patches we provided that were written in Rust mitigated all **11** of these known CVEs, and *(ii)* when providing a Rust-based patch, the number of ROP gadgets present in the binary was reduced compared to native patching. ROP gadgets are machine instruction sequences that an attacker can execute in order to hijack program control flow. Chaining these gadgets together allows an attacker to perform arbitrary operations, hence the importance in minimizing their count as much as possible.

TABLE V
NUMBER OF RESULTING CRASHES FROM TESTS RUN ON SOLDER-PATCHED BINARIES. NEW ERRORS ARE BUGS THAT WERE NOT TRIGGERED WITH TESTS PRIOR TO PATCHING.

Program	#Tests Run	#Errors Found (#New Errors)
OpenSSL	243	0 (0)
OpenSSH	73	0 (0)
Apache	692	0 (0)
Mongoose	30	0 (0)
Mosquitto	188	0 (0)

TABLE VI
AMOUNT OF COVERAGE PROVIDED BY KLEE ACROSS PATCHES FOR EACH CODEBASE AND THE RESPECTIVE NUMBER OF ERRORS FOUND.

Program (#Patches)	Instruction Coverage	#Errors Found (#New Errors)
OpenSSL (2)	73.61%	0 (0)
OpenSSH (3)	95.00%	0 (0)
Apache (2)	87.59%	0 (0)
Mongoose (3)	83.05%	1 (0)
Mosquitto (1)	81.97%	0 (0)

While our goal is not to just reduce the number of ROP gadgets present in a binary, this added benefit is helpful for reducing the attack surface of our targets. Figure 4 shows the total number of ROP gadgets present for each binary where: *DS(Rust)* is where we replace the vulnerable function with a direct Rust translation, *Patch(Rust)* is where we implement a Rust-based patch, *NP* is a native-implemented patch, and *UM* is the unmodified, vulnerable project. In all of the above cases, the SOLDER-generated binaries have fewer ROP gadgets present, even when we only provide a direct translation of a given function and do not patch a vulnerability. Overall, we observe that SOLDER reduces ROP gadget count by an additional **3.7%** compared to the native patches, on average. This shows the feasibility of SOLDER being used over time to further harden a project via its patch.

E. Case Studies

We have shown at a high-level how SOLDER works in Section III. Next, we will provide a case study for one of the evaluated CVEs as an example to illustrate how SOLDER works in more detail.

1) `OpenSSL` (CVE-2021-23841): The OpenSSL public API function `X509_issuer_and_serial_hash` attempts to create a unique hash value based on the issuer

TABLE VII
CHARACTERIZATION OF CVES.

Program	CVE	App Name	CVE Type
OpenSSL	CVE-2021-23841	OpenSSL v≤1.1.1i	Remote DoS
	CVE-2021-3712	OpenSSL v1.1.1-1.1.1k	Remote DoS
OpenSSH	CVE-2016-6515	sshd v< 7.3	Remote DoS
	CVE-2021-41617	sshd v6.2-8.8	Privilege escalation
	CVE-2021-28041	ssh-agent v<8.5	Double free
Apache	CVE-2021-39275	httpd v≤ 2.4.48	Buffer over-write
	CVE-2021-40438	httpd v≤ 2.4.48	Malicious request forwarding
Mongoose	CVE-2021-26530	HTTPS Server	Remote OOB write attack
	CVE-2019-19307	MQTT v6.16	Remote DoS
	CVE-2019-13503	MQTT v6.15	Heap-based buffer over-read
Mosquitto	CVE-2019-11779	Broker v1.5.0-1.6.5	Stack overflow

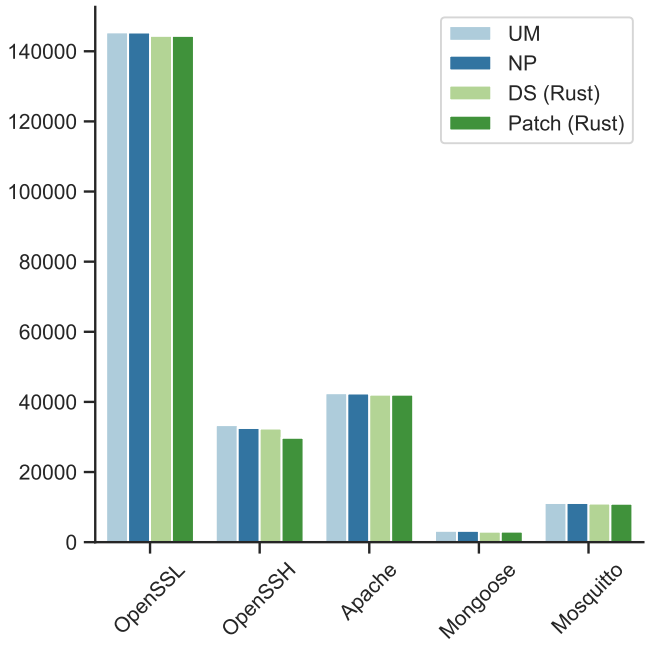

Fig. 4. ROP gadget count for target code base with a direct function swap in Rust (DS (Rust)), a patch implemented in Rust (Patch (Rust)), a native patch in the original language (NP), and the unmodified project (UM).

and serial number data contained within an X509 certificate. However, it fails to correctly handle any errors that may occur while parsing the issuer field. This may result in a crash leasing to a potential denial of service.

SOLDER was able to correctly locate the vulnerable function, `X509_issuer_and_serial_hash`, by getting the patch file that corresponds to the CVE and finding the function that was modified. Next, SOLDER takes the whole function and translates it to Rust code using the tool C2Rust. This produces a Rust-based patch which may make use of some unsafe code features in favor of a fully-automated workflow. The Rust patch is then compiled to LLVM bitcode and tested using the symbolic executor, KLEE. Next, the `name-mangler` pass is run against the patch bitcode to guarantee there are no symbol naming collisions before linking it with the whole program OpenSSL bitcode using `llvm-link`. Now, the `injector-pass` is able to replace the original function with a Rust version.

Finally, we replay any available tests against the patched binary. In this case, OpenSSL comes bundled with very thorough unit tests, which we consider to be sufficient for positing that our cross-language patch does not introduce any new bugs. We also test using a proof of concept exploit against the now patched binary, and verify that it is no longer exploitable (see Section V-D).

VI. DISCUSSION

A. Patching Techniques

We show that SOLDER is capable of patching applications by replacing individual functions with updated implementations. This prototype of SOLDER targets individual functions primarily due to the typical localized nature of software bugs. Other potential options could be to keep the patch separate from the program, and do the patching in-memory, or to adopt a full binary rewriting approach for patch injection. We chose to implement an approach where SOLDER can perform patching and source-level modification operating on LLVM IR, which we believe provides the most general framework for extending support to other languages.

B. Generalizability

In order to demonstrate the applicability of SOLDER beyond C/C++ codebases with patches in Rust, we would need to extend support by writing LLVM frontends for a target language. While we chose Rust to show cross-language support for patching, SOLDER can be easily adapted to support other languages: As long as the language has an LLVM frontend so it can generate bitcode, it will work within our system. Although not included in our evaluation, we conducted tests to show that patches could have been written in Haskell or Go instead of Rust. The only caveat here is that SOLDER requires its LLVM version be consistent with the LLVM version used to generate the project bitcode. In our cases, this is not an issue since our rustc compiler uses LLVM version 12.0, which is the same as the SOLDER system. When extending support to Haskell, however, we are more limited with the capabilities of SOLDER's LLVM passes as GHC (v8.6.5) only supports LLVM version 6.0 for its bitcode output.

C. Limitations

When applying SOLDER to binaries without access to source, we still require knowing the names of the target functions that we wish to replace. This can be infeasible if the binary in question has been stripped of debugging symbols. While we do not have a generalized solution to this application scenario, we leave this reverse engineering problem to our future work. Lifting a binary to LLVM IR is also a non-trivial problem in general, as binary lifting is not decidable and may lead to inconsistencies in support for analysis tasks [39].

Another limitation of SOLDER is that our patch testing phase is built on top of KLEE, which means for full granularity of results, we need LLVM bitcode of any library functions that we want modeled. In our case, that meant manually compiling the Rust standard library to LLVM bitcode so we could symbolically execute the patch using KLEE. This is only required when we want to support new languages like Rust. When writing patches in C++, for instance, we simply use KLEE's version of uClibc [40]. In the case of more complex functions, KLEE may need to limit parameters such as loop unrolling depth. This means that the coverage is not fully complete. We acknowledge that this means we cannot truly prove the absence of bugs, but we are instead providing a level of confidence that we do not introduce any new bugs.

VII. RELATED WORK

a) Retrofitting Legacy Code: Necula et al. [41] presented a program analysis and transformation system called CCured to add memory safety guarantees to legacy C programs. It works by first analyzing the program then attempts to find the safe portions that already adhere to a strong type system with the remainder of the program being instrumented with runtime checks for memory safety. While this provides a good aid for debugging since it enforces memory safety, it does not support actual patching as it only provides instrumentation, and it is only concerned with memory safety enforcement.

Ganapathy et al. [42] presented a mechanism to assist the process of retrofitting legacy code for authorization policy enhancement. It consists of two tools, named AID and ARM. This mechanism combines static analysis and dynamic analysis to identify security-sensitive operations by analyzing canonical code-patterns being executed by the server, and then it instruments these operations and mediates them by adding calls to a reference monitor which encapsulates an authorization policy. Although AID and ARM build an automated system to retrofit legacy code, it does not fix original vulnerable security-sensitive operations, and it can only work on C source code.

Parekh et al. [43] proposed a tool to retrofit autonomic computing onto stovepipe legacy systems, called Kinesthetics eXtreme (KX). KX is a middleware-like infrastructure that provides three components, so as to add autonomic services to legacy systems by keeping monitoring and analyzing systems, and performing reconfiguration and repair. While it can retrofit autonomic capability for legacy systems, it needs to add sufficient sensors around and into systems for monitoring, and the system also needs to provide enough interfaces to allow reconfiguration. Compared with our tool, KX focuses on building an autonomic system that is self-repairing and self-configuring, while SOLDER targets currently existing vulnerable functions.

b) Binary Patching: The challenging task of binary patching has been extensively studied [15], [24], [44]. For example, BinSurgeon [45], and AutoFix-E [46] allow users to write patches using templates or source code annotations. Duan, et al. [47] proposed OSSPATCHER, which patches vulnerable open source mobile applications with source patches. While this prototype provides a layered pipeline that builds function-level binary patches from source code, much like SOLDER; a crucial difference is that it performs the patching in-memory. This means that instead of actually replacing the vulnerable code, they are jumping over it at runtime and running a patched function in place of the original while leaving the vulnerable function still present in the binary. Hu et al. [24] proposed BinPatch, which locates a defective function in a binary and replaces it with a correct version. While similar to SOLDER, they rely on IDA Pro for finding the function entry point and then add an extra `.text` section which connects the patch to the original code via long jumps.

Other tools work assuming that patches are available publicly or from the analyst [48]–[50]. Binary-rewriting [51]–[54] and hot-patching at runtime [48], [49] are also viable patching techniques; however, we focus on precisely targeting individual functions that can be patched to mitigate a vulnerability to minimize changes to the application.

c) Code Reuse Attack Mitigations: In a survey of code-reuse attacks (CRAs) by El-Zoghby and Azer [55], the authors presented different classes of CRA attacks and their mitigations. Among these defenses were: control-flow integrity (CFI) [56], memory layout randomization [57], instruction rewriting [58], and heuristic defenses [59], [60]. These defenses rely on randomizing memory formats, changing instructions of the vulnerable application, or using heuristic thresholds to detect abnormal behavior of software. While these methods are widely adopted, they are general defenses and do not precisely target the vulnerable parts of code. Instead, they add on layers to make exploitation harder while leaving potentially vulnerable code in place.

Kayaalp et al. [61] proposed a mechanism called *branch regulation* which is a hardware-supported protection against code reuse attacks meant to address the limitations of software CFI. This mechanism enforces control flow rules in hardware at the function level to disallow arbitrary control flow transfers. This system relies on annotating functions via binary rewriting then checks any control altering instructions to verify that they target legitimate destinations. While this is a useful alternative to traditional CFI, it does not allow for the replacement of potentially buggy functions, instead opting to enforce checks for arbitrary jumps.

Gionta et al. [62] presented KHide, a system for preventing code reuse attacks through the use of software diversity on kernel code at compile time. KHide is meant to prevent

the use of a priori knowledge of gadget locations to launch attacks. While KHide generates unique kernel images per compilation, the randomization is based around the insertion of NOP instructions and function permutation as shown by Homescu et al. [63] and Fu et al. [64].

Software diversification is a popular moving target defense strategy to prevent code reuse-style attacks. Jackson et al. [65] proposed a compiler-based diversification technique which uses different variation techniques, such as instruction set and register randomization, to create diversity and generate a large number of software variants. Similarly, Franz [66] presented an idea for massive-scale software diversity (MSSD) by generating a unique version of the software for each client which downloads it. Cabutto et al. [67] remove chunks of an executable binary before deployment, and store them on a remote trusted server. Wu et al. [68] present LLVM-based binary software randomization, which apply a number of IR-level transformations (e.g., instruction replacement) prior to compilation. Collberg et al. [69] use a trusted server to continuously and automatically generate diverse code variants, which are then dynamically installed within running clients. Cui and Stolfo [70] propose a host-based defense mechanism called Symbiotic Embedded Machines (SEM). They inject SEMs into host software to provide monitoring and defense services. SEM is an extra component and is executed alongside the host software. Pappas et al. [71] propose in-place code randomization, which breaks the semantics of gadgets used in return-oriented programming attacks. Although SOLDER is a tool meant for maintenance and patching of legacy code, it is still very capable of performing software diversification tasks such as these as well. By replacing arbitrary functions with functionally-equivalent counterparts in another language, we are implicitly introducing diversity into our codebase.

In contrast to existing work, SOLDER can work under the assumption that a patch is already available or will be provided by the analyst. Other tools also work by performing in-memory patching, and do not actively change or remove the underlying vulnerable code. SOLDER, however, fully replaces the underlying vulnerable function with a new implementation and recompiles the binary without the need for annotations or instrumentation.

VIII. CONCLUSION

In this paper, we described SOLDER, a generalized framework to help and enable application developers and third-party users to quickly patch public vulnerabilities in applications without needing to wait for an officially-released patch. As SOLDER is LLVM-based, it is language agnostic so long as there is an LLVM frontend to generate bitcode.

By automating program analysis and transformations necessary for swapping in patches, SOLDER provides an important foundation for program patching and refactoring. As we demonstrated, SOLDER provides a significant utility as a code refactoring tool where users can update legacy codebases with a newer, safer language. While this would not act as security patch, it still helps to reduce the overall attack surface through updating the implementation language. To the best of our knowledge, ours is the first approach to propose function-level patching of software with cross-language patches.

IX. AVAILABILITY

We publicly released the code of our SOLDER prototype on GitHub. The code can be accessed at: https://github.com/solder-project/Solder.

X. ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for their valuable feedback. This research has been partially-supported by NSF Grant 2127200.

REFERENCES

[1] S. Frei, *Security econometrics: The dynamics of (in) security*. ETH Zurich, 2009, vol. 93.

[2] "Average time to fix critical cybersecurity vulnerabilities is 205 days: report," Nov. 2021. [Online]. Available: https://www.zdnet.com/article/average-time-to-fix-critical-cybersecurity-vulnerabilities-is-205-days\-report/

[3] T. Llanso and M. McNeil, "Estimating software vulnerability counts in the context of cyber risk assessments," in *Proceedings of the 51st Hawaii International Conference on System Sciences*, 2018.

[4] "Zero Days, Thousands of Nights," Nov. 2021. [Online]. Available: https://www.rand.org/pubs/research_reports/RR1751.html

[5] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 conference on internet measurement conference*, 2014, pp. 475–488.

[6] D. Moore, C. Shannon, and K. Claffy, "Code-red: a case study on the spread and victims of an internet worm," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002, pp. 273–284.

[7] E. Rescorla, "Security holes... who cares?" in *USENIX Security Symposium*. Citeseer, 2003, pp. 75–90.

[8] "A Guide to Porting C/C++ to Rust," Nov. 2021. [Online]. Available: https://locka99.gitbooks.io/a-guide-to-porting-c-to-rust/content/

[9] M. Miller, "Trends and challenges in the vulnerability mitigation landscape," *USENIX Association*, 2019.

[10] "An update on Memory Safety in Chrome," Nov. 2021. [Online]. Available: https://security.googleblog.com/2021/09/an-update-on-memory-safety-in-chrome.html

[11] L. Clark, "The whole web at maximum fps: How webrender gets rid of jank," Oct. 2017. [Online]. Available: https://hacks.mozilla.org/2017/10/the-whole-web-at-maximum-fps-how-webrender-gets-rid-of-jank/

[12] "A brief history of Rust at Facebook," Nov. 2021. [Online]. Available: https://engineering.fb.com/2021/04/29/developer-tools/rust/

[13] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[14] H. M. Sneed, "Migration of procedurally oriented cobol programs in an object-oriented architecture," in *Proceedings Conference on Software Maintenance 1992*. IEEE Computer Society, 1992, pp. 105–106.

[15] X. Zhang, Y. Zhang, J. Li, Y. Hu, H. Li, and D. Gu, "Embroidery: Patching vulnerable binary code of fragmentized android devices," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 47–57.

[16] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561.

[17] T. Fraser, L. Badger, and M. Feldman, "Hardening cots software with generic software wrappers," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2. IEEE, 2000, pp. 323–337.

[18] W. Venema, "Tcp wrapper," in *UNIX Security Symposium III: proceedings: Baltimore, MD, September 14-16*, 1992, p. 85.

[19] F. M. Avolio, M. J. Ranum, and M. Glenwood, "A network perimeter with secure external access," in *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, 1994, pp. 109–119.

[20] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer *et al.*, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the 1996 USENIX Security Symposium*, vol. 19. USENIX Association Berkeley, CA, 1996.

[21] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[22] V. Ganapathy, T. Jaeger, and S. Jha, "Retrofitting legacy code for authorization policy enforcement," in *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 2006, pp. 15–pp.

[23] "Did Microsoft Just Manually Patch Their Equation Editor Executable? Why Yes, Yes They Did." Nov. 2021. [Online]. Available: https://blog.0patch.com/2017/11/did-microsoft-just-manually-patch-their.html

[24] Y. Hu, Y. Zhang, and D. Gu, "Automatically patching vulnerabilities of binary programs via code transfer from correct versions," *IEEE Access*, vol. 7, pp. 28 170–28 184, 2019.

[25] "Migrate c code to rust," Nov. 2021. [Online]. Available: https://c2rust.com/manual/

[26] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[27] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 745–761.

[28] "A wrapper script to build whole-program llvm bitcode files," Jan. 2022. [Online]. Available: https://github.com/travitch/whole-program-llvm

[29] "Retdec is a retargetable machine-code decompiler based on llvm." Nov. 2021. [Online]. Available: https://retdec.com/

[30] "Bear is a tool that generates a compilation database for clang tooling." Nov. 2021. [Online]. Available: https://github.com/rizsotto/Bear

[31] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[32] "cve-search - a tool to perform local searches for known vulnerabilities," Nov. 2021. [Online]. Available: https://github.com/cve-search/cve-search

[33] "Ropgadget tool," Jan. 2022. [Online]. Available: https://github.com/JonathanSalwan/ROPgadget

[34] "Openssl cryptography and ssl/tls toolkit," Nov. 2021. [Online]. Available: https://www.openssl.org/

[35] "Portable openssh," Nov. 2021. [Online]. Available: https://github.com/openssh/openssh-portable

[36] "Apache http server," Nov. 2021. [Online]. Available: https://github.com/apache/httpd

[37] "Embedded web server," Nov. 2021. [Online]. Available: https://mongoose.ws/

[38] "Eclipse Mosquitto," Jan. 2020. [Online]. Available: https://mosquitto.org/

[39] Z. Liu, Y. Yuan, S. Wang, and Y. Bao, "Sok: Demystifying binary lifters through the lens of downstream applications," in *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA*, 2022, pp. 453–472.

[40] "Klee's version of uclibc," Jan. 2022. [Online]. Available: https://github.com/klee/klee-uclibc

[41] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[42] V. Ganapathy, "Retrofitting legacy code for authorization policy enforcement," in *In Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[43] J. Parekh, G. Kaiser, P. Gross, and G. Valetto, "Retrofitting autonomic capabilities onto legacy systems," 2006.

[44] A. Heinricher, R. Williams, A. Klingbeil, and A. Jordan, "Weldr: fusing binaries for simplified analysis," in *Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, 2021, pp. 25–30.

[45] S. E. Friedman and D. J. Musliner, "Automatically repairing stripped executables with cfg microsurgery," in *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*. IEEE, 2015, pp. 102–107.

[46] 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 19th international symposium on Software testing and analysis*, 2010, pp. 61–72.

[47] R. Duan, A. Bijlani, Y. Ji, O. Alrawi, Y. Xiong, M. Ike, B. Saltaformaggio, and W. Lee, "Automating patching of vulnerable open-source software versions in application binaries." in *NDSS*, 2019.

[48] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 187–198.

[49] Y. Chen, Y. Zhang, Z. Wang, L. Xia, C. Bao, and T. Wei, "Adaptive android kernel live patching," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1253–1270.

[50] "kpatch," Oct. 2021. [Online]. Available: https://github.com/dynup/kpatch

[51] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again." in *NDSS*, 2017.

[52] J. Xie, X. Fu, X. Du, B. Luo, and M. Guizani, "Autopatchdroid: A framework for patching inter-app vulnerabilities in android application," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–6.

[53] P.-A. Arras, A. Andronidis, L. Pina, K. Mituzas, Q. Shu, D. Grumberg, and C. Cadar, "Sabre: load-time selective binary rewriting," *International Journal on Software Tools for Technology Transfer*, pp. 1–19, 2022.

[54] G. J. Duck, X. Gao, and A. Roychoudhury, "Binary rewriting without control flow recovery," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 151–163.

[55] A. M. El-Zoghby and M. A. Azer, "Survey of code reuse attacks and comparison of mitigation techniques," in *Proceedings of the 2020 9th International Conference on Software and Information Engineering (ICSIE)*, 2020, pp. 88–96.

[56] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 1, pp. 1–40, 2009.

[57] T. PaX, "Pax address space layout randomization (aslr)," *http://pax.grsecurity. net/docs/aslr. txt*, 2003.

[58] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 601–615.

[59] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent {ROP} exploit mitigation using indirect branch tracing," in *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, 2013, pp. 447–462.

[60] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

[61] M. Kayaalp, M. Ozsoy, N. A. Ghazaleh, and D. Ponomarev, "Efficiently securing systems from code reuse attacks," *IEEE Transactions on Computers*, vol. 63, no. 5, pp. 1144–1156, 2012.

[62] J. Gionta, W. Enck, and P. Larsen, "Preventing kernel code-reuse attacks through disclosure resistant code diversification," in *2016 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2016, pp. 189–197.

[63] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–11.

[64] J. Fu, Y. Lin, and X. Zhang, "Code reuse attack mitigation based on function randomization without symbol table," in *2016 IEEE Trustcom/BigDataSE/ISPA*. IEEE, 2016, pp. 394–401.

[65] T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz, *Compiler-Generated Software Diversity*, 08 2011, pp. 77–98.

[66] M. Franz, "E unibus pluram: Massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 New Security Paradigms Workshop*, ser. NSPW '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 7–16. [Online]. Available: https://doi.org/10.1145/1900546.1900550

[67] A. Cabutto, P. Falcarin, B. Abrath, B. Coppens, and B. De Sutter, "Software protection with code mobility," in *Proceedings of the Second ACM Workshop on Moving Target Defense*, ser. MTD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 95–103. [Online]. Available: https://doi.org/10.1145/2808475.2808481

[68] B. Wu, Y. Ma, L. Fan, and F. Qian, "Binary software randomization method based on llvm," *2018 IEEE International Conference of Safety Produce Informatization (IICSPI)*, pp. 808–811, 2018.

[69] C. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 319–328. [Online]. Available: https://doi.org/10.1145/2420950.2420997

[70] A. Cui and S. Stolfo, *Symbiotes and defensive Mutualism: Moving Target Defense*, 08 2011, pp. 99–108.

[71] V. Pappas, M. Polychronakis, and A. Keromytis, "Practical software diversification using in-place code randomization," in *Moving Target Defense*, 2013.