Dilipa: Making Micropatches from Edits to Lifted C

Ben Karel, Henny Sipma, Ricardo Baratto, Michael Gordon
Aarno Labs
{bkarel, hsipma, ricardo, mgordon}@aarno-labs.com

***Abstract*—When source code is unavailable, patching security vulnerabilities in binaries requires scarce reverse engineering expertise and specialized tooling. We present Dilipa, a binary micropatching system that enables users to specify patches as edits to lifted C code. Dilipa operates on an AST-based intermediate representation enriched with provenance metadata linking high-level constructs to underlying binary instructions, registers, and memory locations. A frontend compares the original and edited ASTs to extract minimal patch descriptions, and a backend applies them to the binary via direct instruction replacement or trampolines. By focusing on micropatches, small and localized modifications, our approach keeps binary changes minimal and enables post-patch validation through relational binary analysis, providing evidence that no unintended semantic changes have been introduced. We demonstrate Dilipa on three case studies involving real embedded systems, including input validation, buffer overflow, and race condition bugs.**

I. Introduction

While recompiling patched binaries from modified source code remains the most reliable approach to software maintenance, this method is not always feasible in practice. When products reach end-of-life or vendors cease operations, organizations are often left maintaining critical infrastructure with insecure binaries for which source code is unavailable. This gap between the need for security updates and the availability of conventional patching methods presents a significant operational challenge.

Reverse engineers routinely employ decompilers to understand vulnerable functions within binaries, yet they lack analogous tooling support for making small, localized modifications at the binary level. From an organizational perspective, binary-level remediation thus faces two fundamental challenges: first, skilled reverse engineers are a scarce resource; second, the absence of specialized tooling substantially exacerbates the already high cost of their expertise.

Within the realm of possible modifications that can be performed on a binary, *micropatches* represent an important class of binary edits within the world of software sustainability. We define a micropatch as a modification with a localized implementation at the binary level. Micropatches reflect the reality that security-critical fixes often require only small, targeted modifications, e.g., 91% of security patches modify 10 lines or fewer [1].

This paper introduces Dilipa[1], a binary micropatching system designed to address both the scarcity and cost of relying on reverse engineers to manually modify binaries. Our core novel contribution is to enable the formulation of binary patches via edits to decompiled C code. Users specify the patch as modifications to a high-level lifting of binary semantics into C, and under the hood Dilipa automatically detects the modifications and converts them into binary-level patches. This approach is designed to reduce the need for specialized reverse engineering expertise.

Focusing on micropatches both enables our approach to patch creation and also makes *validation* feasible: rather than placing full trust in the patching system, we propose the use of binary-level validation to certify that no unexpected or undesirable semantic changes have occurred in the patched binary.

Finally, for certain restricted classes of changes, our system enables automated patch generation without human intervention, thereby lowering the barrier to binary-level security remediation.

II. System Overview

The core patching workflow involves the following steps:

1) Dilipa automatically lifts the relevant parts of the target binary and presents them to the user as editable C code.
2) The user edits the high-level C code to represent the desired modifications to the binary. They are able to use variables, both local and global. They can make use of high-level control flow statements, such as conditionals, loops, and early returns, and they can add calls to additional functions in the binary.
3) Dilipa automatically analyzes the modified lifting and extracts the edits performed by the user.
4) Dilipa automatically converts the high-level edits into one or more micropatches, which are then applied to the underlying binary. The assembly for each micropatch's code is generated by feeding carefully constructed C code to an off-the-shelf compiler.

The high-level lifting that forms the core of the user-visible workflow is built on top of PIR, short for Patcher Intermediate

This research was funded, in part, by the Advanced Research Projects Agency for Health (ARPA-H). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Workshop on Binary Analysis Research (BAR) 2026
27 February 2026, San Diego, CA, USA
ISBN 978-1-970672-08-4
https://dx.doi.org/10.14722/bar.2026.23062
www.ndss-symposium.org

[1]The name Dilipa is a shortening of "differential lifted patcher" and also names a genus of butterfly—an animal which boasts a light footprint.

Representation (Section IV). PIR represents an abstract syntax tree (AST) of the parts of the binary that are to be patched, plus additional provenance metadata connecting the user-visible "high level" lifting with the underlying binary. PIR is meant to be generated by specialized binary analysis tools that can perform complex static analysis at the binary level. We have built upon the open-source CodeHawk [2] binary analysis platform to produce PIR consumed by Dilipa.

Prior to this workflow, the user identifies the functions involved in the bug to be patched. In practice, this triage draws on multiple sources of information: crash reports with stack traces, vulnerability disclosure reports, available source code, interactive exploration with decompilers such as Binary Ninja or Ghidra, and study of Dilipa's function liftings. Once the relevant functions have been identified, the user can reason about a patch and make direct modifications to the C code lifted by Dilipa.

Like a compiler, Dilipa is split into a frontend and a backend component. The frontend is in charge of extracting the modifications to the high-level lifting and producing a *patch description file* (PD). Each PD file contains a list of patch *fragments*; each patch fragment can provide high level C code that should be added to a binary, or describe other low-level changes to make, such as altering an existing string constant or replacing existing instructions with no-ops. Fragments that add C code contain additional metadata to describe where in the binary the code should be added, and how it should integrate with the surrounding context.

The backend takes the patch description file and the corresponding binary, and emits a patched binary. Alongside the patched binary, the backend also produces a metadata file describing different aspects of the patch that was implemented. This metadata is meant to enable binary analysis systems to perform verification and validation of the patched binary.

The design of Dilipa takes advantage of the following key insight: *Using binary analysis, it is possible to carefully construct C code which, when given to an off-the-shelf compiler, will result in binary code that can be encapsulated and surgically spliced with minimal disruptions to the target binary*.

A critical benefit of surgically applied micropatches is that they enable precise post-patch validation. Because modifications are small and localized, Dilipa can leverage CodeHawk's abstract interpretation engine [3] to perform relational analysis between the original and patched binaries across ELF structure, functions, and basic blocks. This analysis verifies that patch fragments appear only at their designated locations and characterizes semantic impact by comparing sound invariants inferred for each binary, providing evidence of what changed and what did not (Section VI-C).

III. Case Studies

To illustrate the capabilities of Dilipa we walk through three case studies detailing patches that we have implemented and validated as part of our research. All original binaries in the case studies are stripped ARM32 binaries compiled from C source. Table I summarizes the characteristics of each patch.

TABLE I
CASE STUDY PATCH CHARACTERISTICS.

Case Study	Binary Size	Total Fns	Modified Fns	Tramp.	Bytes Mod.
Medical Device	2.8MB	3260	1	✔	68
IDPS: Buf. Overflow	290KB	530	1†	✔	78
IDPS: Race Cond.	212KB	251	1†	✔	16

†Plus 1 dead function overwritten for trampoline space.

A. Production Medical Device

In CVE-2024-12248 [4], the Contec CMS 8000 Patient Monitor was found to have a serious vulnerability due to lack of validation of input from the network which is eventually used to index into a global array. This can lead to an invalid memory access. An interesting aspect of this case study is that the user has several options about how and where to situate the patch. They can choose to put the fix close to where the network data is parsed, or they can choose to fix the issue close to where the memory access occurs. This showcases a useful property of Dilipa: enabling the expression of patches at a high-level allows users to reason about the *best* patch to implement without being constrained by the low-level details that operating at the binary level entails.

The two possible patches are as follows. If implemented at input parsing time, the patch is located inside an input-processing infinite loop, and upon detection of invalid data it discards the data by continuing to the next loop iteration:

```
  rtn_strtok__1 = strtok(0, ":");
  i_token_1 = atoi(rtn_strtok__1);
+ if (i_token_1 > 64 || i_token_1 < 1) {
+   continue;
+ }
  parsed_data.token_1 = i_token_1;
```

If implemented close to the memory access, the patch is located inside of a switch statement (as lifted from the binary), and upon detection of invalid data it performs an early return:

```
  case 2:
+   if (data->token_1 > 64 || data->token_1 < 1) {
+       return; }
    g_data_array[(data->token_1 - 1)].field0 = 1;
    g_data_array[(data->token_1 - 1)].field_4 =
    data->token_2;
```

The simplicity of these patches backs our assertion that micropatches can provide an effective mechanism for verifiable remediations for security-critical fixes. Free space for the trampoline was created automatically by Dilipa by adding a new ELF header (see Section VI-D).

CodeHawk's validation of the patched binary confirms correctness along two dimensions: first, that the C-level edits were faithfully applied to the binary, by comparing the developer's changes against a lifting of the modified binary; and second, that the patch closes the vulnerability, by analyzing the patched

binary's lifting to prove that the undefined behaviors present in the original program have been eliminated. Further details on this case study, including the full validation results, are available in a companion blog post [5].

The next two case studies are based on binaries, and their bugs, developed by the evaluation team of the DARPA Assured Micropatching (AMP) program.[2] In each of the cases, free space for a trampoline was determined by manual analysis, overwriting a dead function. This is because, unlike the previous case study, the binaries use physical addresses (for flashing) and serve as firmware images where the ELF metadata is effectively read-only. In such cases, Dilipa cannot safely modify the ELF structure to create free space (Section VI-D).

B. GridIDPS: Buffer Overflow

An embedded Intrusion Detection and Prevention System (IDPS) has been developed which enables configuration through a USB serial console interface. After deployment, it was discovered that this interface is susceptible to a buffer overflow that can crash the system. The traditional fix for a buffer overflow is to add a bounds check and trigger error recovery if an invalid memory operation is detected. In this particular system, error recovery entails consuming all available data in the serial console, otherwise the system will be left in an unstable state. The following listing contains the patch that was implemented:

```
if (bufferPosition >= 18 || bufferPosition < 0) {
    bufferPosition = 0;
    while (usb_serial_available() > 0) {
        usb_serial_getchar();
    }
    return;
}
```

This case study showcases several aspects of Dilipa:

- `bufferPosition` is a global variable which later on in the function is used as an index into a global array holding input data, the source of the buffer overflow. The patch writer was able to reference it directly without having to worry about loading it into a register or similar low-level concerns.
- An unbounded loop was added that calls functions within the binary to check for available data and consume it.
- Finally, an early return was added to guarantee that the unsafe data was not processed.

CodeHawk's validation of the patched binary was similar to the previous case study, the lifting comparison validates the changes to the modified binary, and CodeHawk verifies that the changes to the liftings remove undefined behaviors present in the original.

C. GridIDPS: Race Condition

Another version of the IDPS binary contains a race condition between the Interrupt Service Routine (ISR) and the main processing loop, described in detail in [6]. The ISR handles incoming CAN bus data and shares the variables `pending` and `processed` with the `get_bitchunk()` function. When the ISR fires between the loads of these two variables, it can reset both, causing `get_bitchunk()` to operate on a stale value of `processed` with a reset `pending`. This leads to incorrect interpretation of CAN frame data.

The fix wraps the two loads in an interrupt-disable/enable pair, ensuring atomicity:

```
+   asm("CPSID i");
    vR2_2 = pending;
    vR3_3 = processed;
+   asm("CPSIE i");
```

The resulting binary patch is minimal: the two load instructions are moved into a single trampoline that surrounds them with interrupt disable/enable instructions. This is an optimized, special-case patch pathway which uses one trampoline instead of two when two adjacent inserted statements are constrained to need identical hook placement. Interrupts are disabled for only two instructions, preserving the system's real-time responsiveness. CodeHawk's relational analysis of the patched binary confirmed that 523 of 526 invariants were retained, with the three lost invariants corresponding solely to the instructions overwritten by the trampoline hook.

IV. PIR

As previously mentioned, the original lifting is presented to the user as plain C code, but is represented internally as an abstract syntax tree (AST) enriched with additional provenance metadata connecting the user-visible "high level" lifting with the underlying binary. The AST design is based on the data structures defined by the CIL project [7]. We refer to the metadata-enriched AST as the Patcher Intermediate Representation (PIR).

PIR may contain additional ASTs of C code which capture the behavior of the binary but at lower levels of abstraction. For example, they may use gotos instead of structured control flow, and may name variables based on platform registers instead of using human-friendly names. Lower level ASTs may also reflect the binary's internal behavior more literally, such as by featuring multiple statements to materialize a string constant.

Provenance metadata links AST nodes between different levels, and also ties AST nodes to the underlying binary. Provenance consists of the following relationships:

- instruction mapping (one to many)
- expression mapping (one to one)
- lval mapping (one to one)
- reaching definitions (expr to instrs)
- flag reaching definitions (expr to instrs)
- definitions used (lval-id to instrs)

Additional forms of metadata include storage, available expressions, and span information. Storage metadata links lvalues to concrete storage locations, such as registers, stack slots, globals, heap offsets, or processor flags. Available expressions provide, for every instruction address, a list of the expressions

[2]Unfortunately, as of the time of publication, these case studies are not publicly available.

available in lvalues—particularly in machine registers. The patcher uses available expressions to help minimize the size of patches it creates by reusing existing values as much as possible. Lastly, span information provides virtual addresses and sizes for instructions and some expressions. Expression spans are intended to provide a connection from AST to assembly code for those assembly instructions that do not have corresponding AST instructions, in particular, return statements and branches.

Instruction and expression mappings may link nodes between different AST levels. Mappings may also provide links to "orphan" nodes that are not rooted in any AST (except via provenance links). This provides the flexibility to specify semantic clarifications that do not exactly map to the structure needed by any given AST level.

Following the structure of C, ASTs consist of a global symbol table, with type definitions, global variables, and function signatures, plus a list of functions. Each function consists of name, virtual address, root nodes for the body (at different levels), plus provenance information.

Although we call it an abstract syntax tree, PIR nodes may form a DAG: node sharing is permitted but not required.

The PIR file is represented concretely in JSON format, including all ASTs and associated metadata.

V. Patcher Frontend

The front end of the patcher takes as input an enriched C lifting along with edited C code, and outputs a patch description file. The edited C code is represented in the same AST format as used in PIR.

A. Identifying DiffSites

The original C lifting is constructed so as to preserve a round-trip property: taking the AST form, converting it to C code, then parsing and reconstructing an AST from the textual C code should produce an AST with structure matching that of the original. It is not necessary that the ASTs be identical. Trivially, node identifiers need not match up. Also, for example, an integer constant may be expressed in hexadecimal format on one side and base-10 notation on the other; this is permissible so long as the value itself is identical.

The round-trip property ensures that any structural differences found between (the ASTs of) original and modified liftings are the result of edits made by the user. Thus, patch extraction amounts to recursively exploring the two ASTs in lockstep, looking for non-trivial divergences. This occurs for each function. Differences in the set of functions within each binary would be handled separately. Currently the patch extraction subcomponent supports the insertion of new functions, but not the deletion or wholesale replacement of existing functions. Alterations to elements of the global symbol table, such as changing the sizes or types of global variables, are also not yet implemented.

A divergence between the two ASTs is recorded as a DiffSite, containing a path through the ASTs along with extra information about edits to statements. For expression-level changes there will be an old and a new expression as the terminus of the path. A DiffSite for a statement-level change may omit either old or new elements, corresponding to deletions or insertions, respectively. We do not wish to fall back to the coarse granularity of recompiling an entire parent block for such changes.

Each DiffSite must be resolved to one or more spans in the binary where the corresponding changes will be made. The mapping is not one-to-one because a single high-level construct may arise from multiple lower level pieces. For example, a large constant may be materialized using a pair of instructions to separately load the top and bottom halves of a register. In that case, the DiffSite for a changed constant would give rise to two changed spans, one for each instruction.

B. From DiffSite to Patch Fragment

From each DiffSite, the patcher synthesizes a list of patch fragments, which form the core of the patch description file. Each patch fragment specifies a patch kind (inserted statements use a trampoline; edited statements or expressions can be overwritten in place, if the new code fits), the new code to use (as textual C), and placement metadata (Section V-E).

Trampoline metadata is also part of the patch fragment, and includes which instructions the trampoline hook should overwrite, if one is needed. This metadata is included even for replacement patches, because there is no guarantee that the compiled code will fit, so fallback to a trampoline must be accounted for. The backend cannot derive this information because it runs without access to PIR, which in turn implies it does not know about instruction boundaries or branch targets: Given a binary, instruction boundaries are relatively easy to obtain using standard disassemblers, but branch targets require deeper analysis, and the design of Dilipa shifts as much analysis as possible to the producer of the PIR file. The full details of our trampoline implementation can be found in Section VI-B.

The patch fragment also provides a logical execution address, which may not be the same as the address of the hook. This situation can occur, for example, when the user's code should execute in between two two-byte instructions. This can be accomplished by overwriting the two with a four-byte hook, and having the logical execution address be two bytes after the hook's address.

Patch fragments also specify which existing instructions should be preserved by copying into the generated trampoline, and whether they should be executed before or after the new code. The logical execution address can be derived from the preserved span information but including both enables a crude consistency check.

The code of each patch fragment is derived from a serialization of the divergent node identified in the DiffSite.

- For nodes with a known destination (e.g., an expression E held in register R), the generated code takes the form `R = E;`.

- Deleted code synthesizes a `nop` or jump instruction as appropriate for the control flow context.
- Edits to conditional expressions generate conditional `goto` statements. The `goto` targets a synthetic label which embeds the relative offset used by the conditional branch in the original binary. The backend then edits the assembly generated by the C compiler to replace the specially-named label with the corresponding relative offset.
- When the callee of a function call is changed without edits to the arguments, the patcher serializes the edited code in a "callee-only" mode which omits the arguments. This is done under the assumption that code generation for a bare function call will produce a call instruction that can be reliably written over the original. This technique works only when the old and new callees share the same calling convention.
- If none of the above cases apply—such as for simple insertions of new statements—the code of the patch fragment is simply the serialization of the divergent node, which is equivalent to the code typed by the user modulo canonicalization performed by CIL.

Each patch fragment also includes a list of symbol metadata, which provides a concrete basis for the symbols referenced from the user's code. The symbol information is used by the patcher backend to replace symbolic named references with concrete snippets that access the appropriate global variable, register, virtual address, or other entity.

Finally, to support the user inserting a new `return` statement, trampoline patch fragments may also include a *return sequence*—a snippet of assembly code, customized for the chosen location, that effects a return from the function being modified. A return sequence typically consists of a stack adjustment followed by a return instruction, and is derived from PIR analysis.

Our original design simply identified the addresses of existing return sequences to reuse, but we found this approach insufficiently robust. Usable existing return sequences may not exist for particular program points for two reasons. First, a temporary `alloca` may change the stack size in a limited region, with cleanup code that is separate from any extant return sequence. Second, on platforms such as ARM, a function may only have a conditionally-executed return sequence. To use such a sequence, we would need to synthesize appropriate processor flag settings before jumping to it. Synthesizing our own return sequences based on the static analysis already being performed proved more robust.

C. Edit Recognition

The patcher has a fair amount of discretion in how it maps changes on liftings to binary patches. Example: deletion of the `else` block of an `if` statement could be implemented by altering the targets of any jumps from the conditional test, or by overwriting the entry block of the `else` with an unconditional jump to the statement following the `if`. (This assumes there is such a statement; if both branches of the `if` end with a `return` statement, deleting the `else` block essentially introduces undefined behavior, and like the C compiler itself, the patcher makes no promises about the consequences of doing so.)

Combining edits introduces more degrees of freedom. For example, suppose the user takes an existing block of statements, adds a conditional `if` wrapper around them, and also deletes a function call from the block. The patcher could treat this as two specialized edits (one using a trampoline to evaluate the conditional of the `if`, and one to overwrite the function call with a no-op instruction), or as one combined edit doing a wholesale replacement of compound statements.

In these cases, more sophisticated edit recognition during patch extraction helps minimize the binary-level changes. There are also forms of edits which the patcher does not recognize, such as changing the declared type of a variable. For some changes, such as altering the signedness of a variable, equivalent effects can be produced by e.g. inserting casts at comparison sites.

D. Supported Edits

Edits to calls, stores, and statements are handled separately from edits to expressions. Deletions are the simplest case.

Deletions: An instruction (that is, a call or assignment) may be deleted by overwriting it with a nop. Other statements may be "deleted" with an unconditional jump.

Insertions: An inserted statement will generally have a range of possible spans it could target: the new statement must execute after what came before it, and before what comes after it. Those boundaries may be other statements, or may be derived from the underlying control flow graph, such as for insertions at the start of a conditional block. In the general case, the boundary may not exist, such as for code inserted after a return statement, or it may be ill-defined, such as for code inserted at the start of a function, where the code may or may not need to come after the function's stack frame has been constructed. For more details on how the patcher picks one placement location from a range of possibilities, see Section V-E.

Labeling Existing Statements: The patcher also supports adding new label statements to serve as the targets of inserted goto statements. The span associated with an inserted label is more constrained than for a normal executable statement, because we must be careful not to skip any setup code needed for the target statement to execute correctly. For inserted executable statements, placing the hook later will not result in any earlier instructions being skipped. Section VI-E discusses in more detail some of the limits of the patcher's approach.

Edits to Expressions: While statements have an intuitive mapping with the instructions that implement them, expressions can be more subtle. The same expression, such as an integer constant, can have different representations depending on context. As a function argument, the integer might map to an instruction that loads a register or a stack slot. When used as a limit within a conditional comparison, the integer value may not have any independent existence within the binary, being encoded as a mere subfield of its parent comparison operator. Thus, the patcher walks up from the divergent expression

node until it finds a node (expression or instruction) with an associated span. Along with the span, the patcher records the destination location, if any, for the expression (currently only registers are supported). Thus, if the original code initialized `r3` with the constant `6`, and the user edits that constant to be `42` instead, the patcher will generate `r3 = 42;`.

Edits in Context: To help produce minimal micropatches, the patcher recognizes edits at finer granularity than strictly necessary. For example, when looking at edits to a function call, the user may have modified one-to-all of the function arguments but left the callee unchanged, or may have modified the callee without altering the function arguments, or may have modified the callee and one-or-more arguments, possibly with a different arity than the original call. The patcher has leeway in these scenarios between producing one or more "surgical" replacements to subcomponents of the call, or simply generating a completely new function call.

E. Placement of Inserted Code

Given a range to consider for an insertion, the patcher must pick the best available location, which will be overwritten with a jump to a trampoline which orchestrates the execution of the new code. The overwritten instruction will be relocated to the trampoline, either before or after the new code, as needed. Some locations are unsuitable because they involve instructions that are hard to arbitrarily relocate (e.g. non-branch, non-call PC-relative instructions). Other locations are unsuitable because they partially overlap with the targets of existing branches, which would be broken when trying to partially execute the inserted hook. Due to these constraints, the patcher may need to insert additional padding bytes around the hook to remain instruction-aligned. For example, consider a sequence of code bytes AAAABBCC for a three-instruction sequence on the variable-length Thumb-2 instruction set, where we want to execute in between B and C, but cannot overwrite C (perhaps because C is the target of a jump). We must instead overwrite the six bytes AAAABB with a hook and two padding bytes, diverting to a trampoline which executes A and B, then our new code. The total size of the hook, jump plus padding, is called its footprint.

Because there are typically only a small handful of locations to consider, the patcher does a brute force search over every possible placement option to construct a list of all acceptable candidates. These candidates are then ranked by the number of subexpressions of the inserted code which can be reused from the available expressions specified in PIR metadata, and the location with the greatest potential for reuse is chosen, with ties broken by smallest footprint. Currently the patcher only considers syntactically identical subexpressions, and does not consider what adjustments it might be able to make to enable more reuse. For instance, suppose the user has changed an integer constant `0` to `x - 1`, at a location where the expression `x + 1` is held in register `r3`. The value of `x - 1` could be implemented as `r3 - 2` rather than by naively re-loading the value of `x` from a global variable—but the patcher does not yet do this.

VI. Patcher Backend

The back end of the patcher takes a patch description and the corresponding binary, and emits a patched binary.

A. Patch Application

Some patches are nearly trivial. The very simplest patches are those which overwrite some existing binary content with new bytes. This can result from every use of a string constant being changed in concert. The next simplest patches are those which insert new strings. The patcher first searches the loaded segments of the binary for an existing string that matches the desired contents. Since C strings are null-terminated, a suffix of an existing string can be used as-is, with a bare address. If the desired string does not already exist in the binary, the patcher goes about acquiring the requisite space; see Section VI-D.

Patches involving code have two main paths through the patcher backend: replacement or insertion via trampoline. The general idea is that trampoline code will be compiled as a function, which will be inserted into the binary as-is and called by a snippet of platform-specific assembly code. Non-trampoline code will be compiled as part of a function augmented with *marker statements*. A marker statement has two purposes: first, it serves as a compiler barrier, ensuring that unrelated code cannot be moved past it. Second, it must produce a single, easily-identifiable assembly instruction, distinct from those produced by the compiled snippet. We can then extract the assembly instructions for our code snippet, without extraneous elements from function preludes, by looking for the markers.

The first step for both insertion and trampoline patches is to perform initial transformations on the incoming code:

- Replacement of string literals with corresponding addresses. String literals within `asm` statements are not subject to replacement.
- Replacement of symbolic names with concrete accessors (for registers, byte indexing with casts, etc).
- Replacement of known builtins with temporary placeholders. Compilers are wont to do things like open-code "primitive" functions such as `memset`. Those tricks work in a standard compilation setting, but interfere with patching.

Code destined for a trampoline then additionally undergoes two further transformations: control flow munging and register redirection. Control flow munging replaces "external" control flow constructs, such as `break;` or `return;` with a status code indicator. The trampoline wrapper code that calls the trampoline body will inspect the status code to determine what kind of control flow effect, if any, should be taken. The other transformation, register redirection, helps ensure a separation between register accesses that the patch code does intentionally, and register accesses that are a byproduct of the compiler's code generation choices. The trampoline wrapper spills all registers to the stack, and passes a pointer to the body. Source to source rewriting of, e.g. `r2` into `spilled[2]` allows intentional register effects to be separated from the compiler's register allocation choices.

The incoming code is placed in a function body with signature `int payload(uintptr_t* spilled)`. It is augmented with declarations for types and globals, thereby producing a valid compilation unit.

For non-trampoline payloads, register references are handled via GCC's local asm register constraints, rather than via register redirection. These ensure that C-level variables are held in the appropriate registers. One caveat is that this guarantee only holds at designated statement boundaries. In between user provided statements, or even within the code for a single statement's subexpressions, there is no enforcement of the intended correspondence. As an example of the latter, given `r2 = 0xa5a5a5a5; r7_plus_12 = r2;` on ARMv7, one might expect a simple two-instruction sequence of `movw ; str`. But depending on the surrounding code and optimization level, the compiler may produce

```
mov.w  r0, #-1515870811
str    r0, [r7, #12]
mov.w  r2, #-1515870811
```

This is valid according to the semantics of C (extended with local register constraints as implemented by GCC and Clang), but is suboptimal for our purposes, since it will unnecessarily require a trampoline. While programmers often think of C as having a direct mapping to assembly, the specific assembly produced is often hard to predict.

The overall form of a non-trampoline patch fragment is a function which first declares register-held variables, then contains a marker instruction (with register constraints), then the input code, then another marker instruction, then label declarations for any `goto` targets mentioned by the input code.

The register-held variables need a way of conveying to the compiler that their contents are unknown but not unused. Declaring an uninitialized stack slot turns uses into undef, which can in turn drive LLVM into detecting undefined behavior and generating a trap instruction. Calling a function can induce spills of the ABI-specified return register. The cleanest option so far appears to be loading values from a constant pointer value. (Since this code occurs outside of the marker instructions, it will not make its way into the final binary; it is only needed to guide the compiler into not misusing registers).

We are also somewhat careful in handling existing register usages. We don't want to clobber "outgoing" registers which contain the results of computations we want to preserve/observe. Likewise, we don't want to clobber any incoming registers which contain e.g. function arguments, because that could induce spills and wind up with codegen using undesirable registers within our framed section. Thus patch fragments may specify a list of "preserved registers".

Given a compilable non-trampoline patch fragment function, we will compile it at multiple optimization levels and see whether any of the resulting instruction sequences fit the space we are replacing into. (Trampolines are always compiled with maximum optimization). If any fit, we generate a pending change to copy the compiled bytes into the binary; otherwise, we re-compile the input code using the trampoline codepath as a fallback option.

When edits to expressions produce code that is too big to overwrite its target, we simply fall back to using a trampoline instead of a direct replacement. The result will not be quite as tight of a patch, but trampolines provide a reliable abstraction to protect inserted code from its environment, and vice versa.

In the rare case that the generated code is both wrong (e.g. because necessary instructions are moved outside of the marker boundaries) and correctly sized (perhaps because other instructions were moved in), we will generate a patch, and patched binary. Our assumption is that preliminary review by CodeHawk will flag the broken patch.

One form of code that is not well supported by this setup is unconditional `goto`s, for which the C compiler is eager to statically evaluate and thus erase all intervening code—including the marker instructions. So we need some other translation of direct jumps. We expect we could handle unconditional jumps by temporarily transforming them into calls for codegen purposes, then substituting a jump instruction in the generated assembly.

B. Trampolines

Trampolines are composed of three parts. The *hook* is an unconditional jump, which overwrites one or more instructions from the function being modified. The hook may require nop padding to remain instruction-aligned. The hook jumps to the *wrapper*, which helps mediate between the execution environment of the target function and the user's code. The wrapper has three jobs: it spills & restores registers, executes any instructions overwritten by the hook, and also performs any control flow requested by the user. The user's code is lightly transformed and fed to the C compiler as a simple function, which we call the *body*.

The signature for the body takes a pointer to an array of spilled register values, and returns a status code indicating the final control flow behavior of the user's code. When the user writes `return;`, we should not copy it as-is into the trampoline body, because we must be able to distinguish explicit early returns from the body (which should subsequently return from the hooked function) from implicit returns, which resume executing the instructions that come after the hook. The body thus returns a distinct status code for each kind of terminal control flow (return, break, continue, fallthrough). The body takes a pointer to spilled registers, which are used to capture writes to concrete machine registers. Otherwise the writes would be lost when the body returns to the wrapper. This scheme also frees us from micromanaging the compiler's use of temporary registers, significantly simplifying the patcher.

When situating multiple trampolines in a single patch, it may occur that the hooks for several trampolines are overlapping or adjacent. In that case, only the first hook is kept, and the wrappers of all but the last are modified to chain directly.

Other trampoline optimizations are also possible. When the body does not require temporary registers, the body can be inlined into the wrapper. When the body does not explicitly

modify any registers, the trampoline can elide the spills and restores from the wrapper. When the body can only return a single value, the wrapper need not do any dynamic dispatching. Combining these optimizations, where applicable, can result in extremely low overhead trampolines, with total overhead of only two jump instructions.

Replacement patches have a trivially provided target span, but trampoline patches must find or create space for the wrapper and the body.

Just as a linker takes object files and fixes up relocations in the process of creating an executable file, so must the patcher take the object code produced by the compiler and alter relocation sites based on provided symbol information.

C. Validation of Patched Binaries

Patch validation seeks to demonstrate that the applied patch has its intended effect and no unintended side effects. CodeHawk's relational analysis takes as input the analysis results of the original binary, the patched binary, and the patch results file produced by the patcher. The analysis proceeds in four stages: (1) differences in ELF structure (segments, sections, virtual addresses) between the original and patched binaries are recorded, and new sections or segments are registered to direct the disassembler to new code regions; (2) the patched binary is disassembled using the patch results file to locate trampolines and payload functions, which are inlined by the disassembler so they can be analyzed in the context of the patched function; (3) control flow graphs for all functions in the patched binary are constructed and syntactically compared with those in the original, identifying which functions changed and whether functions were added or removed; and (4) modified functions are analyzed in depth and compared semantically using invariants generated by CodeHawk's abstract interpretation.

Because payload functions are inlined, invariants downstream of the patch reflect its effect within the function's actual state at the patch point. The semantic comparison serves two complementary purposes. For vulnerability remediation, stronger downstream invariants can establish the absence of the targeted undefined behavior. Conversely, invariants are compared instruction-by-instruction for all referenced variables, including globals, to detect unintended changes such as incorrectly restored or clobbered registers, overly strong guards, or erroneous control flow. Differences are presented to the user for review.

D. ELF Surgery

Inserting new code into an ELF binary is a little less trivial than it might first appear. The ELF file format itself makes it seem easy: with the file header's `e_phoff` field, the procedure should be as simple as 1) append arbitrary new data to the end of the binary; 2) append a new program header table, with a new `LOAD`ed segment; 3) rewrite `e_phoff` with the offset of the new header table. Unfortunately, even as recently as version 5.10.109, released in 2022, the Linux kernel did not support such (valid) ELF files, because it computed the program header virtual address as the binary's load address plus `e_phoff`. Important tools like QEMU have also performed the same broken computation.

To retain compatibility with deployed devices running older Linux kernels, we must position our new program header table within the first loaded segment. Since it requires a new segment entry, the new table is larger than the old table, and cannot overwrite it in place. There are two problems with merely prepending our new table at the start of the binary (along with a copy of the magic number and file header). First, the binary's contents will appear at changed virtual addresses. This can be compensated for by altering the `vaddr` entries for loaded segments, reducing their values by the same number of bytes being prepended to the binary. This ensures that the binary's original contents remain unshifted in virtual address space. It is applicable so long as the new `vaddr` entries remain valid. The second constraint is that segments must be loaded at page-aligned addresses. We address both issues by inserting enough padding after the new segment header table to ensure that the total expansion is a multiple of the page size.

E. Limitations of Assumptions Made During Patching

The interactive patching workflow we've described is predicated on a key assumption: if the PIR is constructed to match the binary's semantics, then propagating PIR-level changes to the binary level will produce a binary with semantics that match the modified PIR.

This assumption is often true, but not always. Consider this counterexample:

```
2e8a:      adds r1, r2, #0x1
2e8c:      strb r0, [r4, r2]
2e8e:      str  r1, [r3]
2e90:      pop {r4,r5,r6,pc}
```

The corresponding lifted statements are:

```
inputBuffer[bufPos] = r0;
bufPos = bufPos + 1;
return;
```

Note that the ordering between high and low levels does not match. The first lifted statement corresponds to the middle `strb`, while the increment of `bufPos` involves both the first and third instructions.

Let us examine what the assembly is doing. Initially, `r2` holds `bufPos`'s loaded value. Register `r1` is loaded with the incremented value. Then the store to `inputBuffer` occurs. Finally, the incremented value of `bufPos` is written to memory, and the function returns.

Now suppose the user wants to insert `if (bufPos == 18) { bufPos = -1; }` between the two lifted statements. How and where should that code be added?

The patcher has a clear, intuitive model for "where": the write of `inputBuffer` is the `strb` and the increment of `bufPos` is the `str`, so the user's code should execute between the two (at address `0x2e8e`).

Now observe: if (because `r2` holds `bufPos`) we compile the user's code into `if (r2 == 18) { r2 = -1; }`, then we go astray;

`r2` is dead and `r1` has already been computed. If we read and write to the global variable from which `bufPos` was loaded, we will likewise obtain the wrong result.

The user reasons about the lifting statement-by-statement, but the execution is more intermixed. This disconnect is one key challenge when micropatching optimized binaries; it erodes the degree to which the patcher can operate in a localized manner. Note that had the original source contained the user's inserted code, the compiler would have produced a different scheduling for the surrounding code.

The patcher makes several reasonable-seeming assumptions that fail to work for this example:

- If the user writes code between lifted statements X and Y, the patch code must execute between the addresses at which X and Y take effect.
- If the user refers to symbol X multiple times, the translation of that symbol should be consistent.
- Inserting new code should not modify the execution of other instructions (modulo relocating them via trampoline hook).
- The user's code should be compiled in a maximally black-box manner; the patcher should not need to perform sophisticated static analysis of the user's code to obtain correct results.

Dropping these assumptions greatly complicates the patcher's task, which is why the patcher makes these assumptions to begin with. Our vision was to have a relatively "dumb" patcher, performing only syntactic transformations itself, with deep semantic analysis offloaded to other tools such as CodeHawk. This example exposes a limitation of this simple division of labor. A mostly-syntactic patcher can produce a large variety of useful patches, but there is a non-trivial tradeoff between the engineering effort invested in automated patching and the degree of automation achievable.

One correct translation of the user's code would be `if (r2 == 17) { r1 = 0; }`. This violates the second and fourth assumptions listed above: the treatment of `bufPos` is inconsistent, and a non-trivial transformation was applied to the user's code to turn the assignment `= -1;` into `= 0;`. Alternatively, we could violate the first and third assumptions by placing our patch before `adds`, but that would also require modifying the `strb` instruction to use the original unmodified value of `bufPos` from `r2`.

Users can detect incorrectly applied patches via careful review of CodeHawk's inferred invariants (see Section VI-C).

F. Direct Synthesis of Patch Descriptions

Extracting patch descriptions from manual edits to C liftings is a powerful workflow for ad-hoc binary modifications. Because patch extraction and application are separate steps, the Dilipa patcher also permits directly synthesizing binary patches for automated hardening of legacy binaries.

G. Commentary

One interesting consequence of our approach is that it is relatively robust to imperfect liftings, especially in comparison with patching based on whole-function recompilation. We essentially only require fidelity in the *modified parts* of a lifting. The primary risk of incorrect liftings is not that it will cause incorrect patches to be produced. Rather, it is that the patch will not be possible to create, because the instructions in the binary which must be altered will be omitted.

On the flip side: while we end up with lax requirements on semantic correctness, we do have relatively stringent requirements on type consistency. The CIL AST, which is the basis of PIR, is designed to capture semantics over syntax, so the AST depends on having consistent type information. Some things which are valid at the binary level, e.g., reusing a register at a different type, are not supported by C's type system, and therefore must be avoided when producing a lifting.

The use of C to represent patches has pros and cons. On the plus side, it is expressive enough to enable many patches, most low-level programmers are familiar with it, and it allows us to get high-quality code generation on a diverse set of platforms, without being tied to a single compiler. On the minus side, there is a fundamental impedance mismatch with the sorts of changes one might care about expressing at the binary level. For example, calls made directly vs those made indirectly via a PLT are not distinguished in C source, but the difference is quite important when doing binary patching. Another example is that global variables in C cannot be assigned explicit virtual addresses, which means that the patcher cannot express reshufflings of globals directly via edits to liftings.

VII. Implementation

The core patching tool is implemented in approximately 10,000 lines of Python, with support for 32-bit ARMv7 and Thumb-2 ISAs. Dilipa is open-sourced and publicly available.[3] We have plans to extend support for 32-bit MIPS, x86, and x86_64. Most architecture-specific code is encapsulated within a 315 line file. The patcher uses Clang and LLVM to compile code snippets, using a custom pass list to avoid constant propagation and dead code elimination. The patcher also makes use of `llvm-mc`, `llvm-objdump`, and `llvm-readelf`.

Our extensions to CodeHawk for generation of PIR required approximately 12,000 lines of OCaml and 24,000 lines of Python. By line counts, about 20% of the OCaml and 33% of the Python extensions are architecture-specific. An additional 4,000 lines of Python implement relational binary analysis.

VIII. Future Work

The patcher currently relies on CodeHawk both to produce PIR for it to consume, and to validate that the patching process did not introduce unintended changes in the binary. In principle each of these connections could be severed. PIR could be produced by other binary analysis tools like IDA Pro, Binary Ninja, or Ghidra (which could likewise host the patching GUI). Much of PIR's metadata, such as reaching definitions, are standard fare in static analysis. However, return sequences are bespoke, and available expressions might be

[3] https://github.com/Aarno-Labs/dilipa-patcher and https://github.com/Aarno-Labs/dilipa-cil.

difficult to replicate, as they are downstream of CodeHawk's sound invariant inference.

There are two obvious ways one might improve upon the capabilities of the patcher design we have laid out: expand the set of supported modifications to liftings, and expand the range of binaries to which any given modification can be applied. For the former, it would be interesting to explore the use of attributes in C to reify some patching-specific concerns, such as annotating global variables with information about their associated virtual addresses and functions with procedure linkage metadata, in the language itself. For the latter, one has several avenues of attack. Expansion of PIR metadata could permit the patcher to create more sophisticated patches, but that comes at a cost in both patcher complexity and static analysis burden. One could also explore intermediate patching strategies (such as recompiling basic blocks) to reduce the need for detailed inter-instruction dependence metadata. We believe that basic block recompilation, in particular, is a promising avenue for handling patches to predicated instructions, which are challenging for our current design to handle.

On the validation side, we are working toward automated checking that all patch components comply with the patch results file, with the goal of showing that the patch description, together with an isolation property, implies the targeted security property.

IX. Related Work

A number of approaches have been proposed to use binary patching in the context of software maintenance. Commercial solutions exist, the most prominent being 0patch [8], which applies binary micropatches for 0-day and unpatched vulnerabilities to running Windows applications. Dilipa automates the bookkeeping required to create such micropatches, but does not treat patches as independently distributable artifacts, and does not address the distribution of patched binaries. Other approaches take a more coarse-grained approach by focusing on function-level remediation: Vicarius [9] blocks vulnerable functions from execution, while CloudLinux's LibCare [10] dynamically replaces vulnerable functions with recompiled versions of patched ones in linux applications.

Academic research has also focused its efforts on the problem. Like Dilipa, PRD/BinrePaiReD [11] focuses on developing patches using a higher-level abstraction than pure binary or assembly code by decompiling select target functions. Unlike Dilipa's focus on micropatches, their patches do wholesale replacement of the modified functions, providing detours so the original vulnerable functions are not called. From first principles, function-granularity recompilation is neither better nor worse than instruction-granularity micropatches. The former is more robust to semantic flaws derived from the interplay between the patch and the existing code, because it replaces the existing code at a pre-existing abstraction boundary. The latter is more robust to lifting errors in unrelated parts of a patched function. IRENE [12] supports micropatches, though not at the granularity of individual assembly instructions, with modifications expressed in a custom domain-specific language.

Egalito [13] provides a mechanism to perform in-place binary modifications without having to modify all memory accesses and offsets as binary code is moved around. This provides an alternative to trampolines as used by Dilipa, and can provide better performance characteristics if a patch is located in a hot path on the binary. However, to be successful it must precisely identify all instructions and code references in the binary, not those in the functions that need to be patched, significantly raising the bar for successful remediation. A key difference from Dilipa is that Egalito lifts binaries to a low-level IR, rather than a human-friendly programming language.

Finally, ARMPatch [14] and Patcherex2 [15] are systems that enable the development of binary patches with a more manual approach. Both require a human operator to specify the type of patch to be enacted, either using high level code to be compiled by the system or low-level instructions to be inserted directly into the binary. Neither one guarantees minimal modifications to the target binary, and in fact, their patches can result in modifications with repercussions across the whole binary (e.g., due to code movement).

Acknowledgments

We thank the anonymous reviewers for their constructive feedback. This research was funded, in part, by the DARPA Assured Micropatching (AMP), contract N6600120C4025, and by the Advanced Research Projects Agency for Health (ARPA-H) under the DIGIHEALS program.

References

[1] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *CCS*, 2017.
[2] "CodeHawk Tool Suite," https://github.com/static-analysis-engineering. [Online]. Available: https://github.com/static-analysis-engineering
[3] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Principles of Programming Languages*, 1977.
[4] MITRE. (2024) Cve-2024-12248. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-12248
[5] Aarno Labs. (2025) High-assurance remediation of CVE-2024-12248. [Online]. Available: https://www.aarno-labs.com/blog/post/high-assurance-remediation-of-cve-2024-12248/
[6] R. Chatterjee, B. Karel, R. Baratto, M. Gordon, and J. Daily, "Assured micropatching of race conditions in legacy real-time embedded systems," in *Real-Time Autonomous Systems Security*, Jul. 2024.
[7] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: intermediate language and tools for analysis and transformation of C programs," in *Compiler Construction*, 2002.
[8] 0patch. [Online]. Available: https://0patch.com/
[9] Vicarius. [Online]. Available: https://www.vicarius.io/
[10] "Cloudlinux libcare." [Online]. Available: https://github.com/cloudlinux/libcare
[11] P. Reiter, H. J. Tay, W. Weimer, A. Doupé, R. Wang, and S. Forrest, "Automatically mitigating vulnerabilities in binary programs via partially recompilable decompilation," *Dependable and Secure Computing*, 2025.
[12] I. Smith, F. Bertolaccini, W. Tan, and M. D. Brown, "IRENE: A toolchain for high-level micropatching through recompilable sub-function regions," in *MILCOM*, 2024.
[13] D. Williams-King, H. Kobayashi, K. Williams-King, G. Patterson, F. Spano, Y. J. Wu, J. Yang, and V. P. Kemerlis, "Egalito: Layout-agnostic binary recompilation," in *ASPLOS*, 2020.
[14] "ARMPatch: A Binary Patching Framework for ARM-based IoT Devices," *Journal of Web Engineering*, vol. 20, no. 6. [Online]. Available: https://journals.riverpublishers.com/index.php/JWE/article/view/7077/9751
[15] Patcherex2. [Online]. Available: https://purseclab.github.io/Patcherex2/