P³: Reasoning about Patches via Product Programs

ARINDAM SHARMA, Imperial College London, United Kingdom
DANIEL SCHEMMEL, Imperial College London, United Kingdom
CRISTIAN CADAR, Imperial College London, United Kingdom

Software systems change on a continuous basis, with each patch prone to introducing new errors and security vulnerabilities. While providing a full functional specification for the program is a notoriously difficult task, writing a *patch specification* that describes the behaviour of the patched version in terms of the unpatched one (e.g., "the post-patch version is a refactoring of the pre-patch one") is often easy. To reason about such specifications, program analysers have to concomitantly analyse the pre- and post-patch software versions.

In this paper, we propose P^3, a framework for automated reasoning about *p*atches via *p*roduct *p*rograms. While product programs have been used before, particularly in a security context, P^3 is the first framework that automatically constructs product programs for a real-world language (namely C), supports diverse and complex patches found in real software, and provides runtime support enabling techniques as varied as greybox fuzzing and symbolic execution to run unmodified.

Our experimental evaluation on a set of complex software patches from the challenging CoREBench suite shows that P^3 can successfully handle intricate code, inter-operate with the widely-used analysers AFL++ and KLEE, and enable reasoning over patch specifications.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Patches, product programs

ACM Reference Format:
Arindam Sharma, Daniel Schemmel, and Cristian Cadar. 2025. P³: Reasoning about Patches via Product Programs. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 367 (October 2025), 27 pages. https://doi.org/10.1145/3763145

1 Introduction

Most programs lack a full specification of their intended behaviour, much less a proof that they conform to it. However, during the continued evolution of programs, it is oftentimes clear to the developers what the intended goal of a patch is. For example, a refactoring should not change the behaviour of the program, and a crash fix should make the program return a specific result on some inputs instead of crashing, with all other behaviour remaining unchanged. Even if the intended behaviour cannot be easily specified, developers can greatly benefit from the construction of inputs that lead to different outputs across versions. Such inputs are often useful while validating a patch, and are prime candidates for inclusion in regression test suites [16].

To reason comprehensively about behavioural changes across versions, program analysers have to reason concomitantly about both the pre- and post-patch software versions. A way of achieving this is via product programs [2, 11]. At a high-level, a product program interleaves the statements from the two constituent versions, renaming variables to ensure no name clashes. For instance, consider this illustrative example where the two program versions are:

Authors' Contact Information: Arindam Sharma, Imperial College London, London, United Kingdom, arindam.sharma@imperial.ac.uk; Daniel Schemmel, Imperial College London, London, United Kingdom, d.schemmel@imperial.ac.uk; Cristian Cadar, Imperial College London, London, United Kingdom, c.cadar@imperial.ac.uk.

ACM 2475-1421/2025/10-ART367
https://doi.org/10.1145/3763145

```
1 // version 1
2 x = y + 1;
3 x = 2 * x;
```

```
1 // version 2
2 x = y + 1;
3 x = x << 1;
```

Then their product program would be:

```
1 x1 = y1 + 1;
2 x2 = y2 + 1;
3
4 x1 = 2 * x1;
5 x2 = x2 << 1;
```

with the variables renamed to have the suffix 1 in the first version and 2 in the second.

With the product program in place, one can easily write a *patch specification* [7] stating, in our example, that the final values of `x` are the same in the two versions:

```
1 assert(x1 == x2);
```

Product programs have been mainly used to reason about security hyper-properties such as non-interference [2]. More recently, product programs and similar constructs have started to be employed for analysing software patches [18, 28]. In particular, our recent vision paper [7] introduces the idea of (partial) patch specifications that describe the behaviour of the patched version in terms of the unpatched one, and shows how these patch specifications can be validated by *manually* constructing product programs. However, as far as we know, there is no framework that can automatically construct executable product programs for real-world software versions. This would allow arbitrary off-the-shelf analysers, such as greybox fuzzers and symbolic execution engines, to comprehensively analyse patch specifications of the form above, or automatically discover inputs that trigger output differences across versions.

In this paper, we introduce P^3, a framework for automated reasoning about *p*atches via *p*roduct *p*rograms. To the best of our knowledge, P^3 is the first framework that automatically constructs product programs for a real-world language (namely C), supports diverse and complex patches found in real software, and provides runtime support that enables techniques as varied as greybox fuzzing and symbolic execution to run unmodified. While a real-world language like C includes many low-level constructs and edge cases, P^3 handles a vast majority of practical scenarios and is designed to be extensible to support additional cases as needed. The accompanying runtime system within P^3 enables unmodified use of existing analysis tools such as greybox fuzzers and symbolic execution engines, and can likewise be extended to accommodate more specialised use cases.

While many other techniques exist for testing code patches, *the key advantage of P^3* is that it enables arbitrary off-the-shelf program analysers to reason about multi-version code and take advantage of patch specifications, effectively becoming *differential* program analysers.

Our experimental evaluation on a set of complex software patches shows that P^3 can successfully handle real-world code, interoperate with the widely-used analysers AFL++ [12] and KLEE [6], and enable reasoning over semantic patch specifications.

In summary, the main contributions of this paper are:

(1) P^3, a technique for constructing the product program of *different* program versions (instead of identical versions, as done when reasoning about hyper-properties) for real-world C programs. P^3 relies on an abstract syntax tree (AST) matching algorithm which is used to generate patch annotations that model cross-version divergences in the product program. Unlike prior work on patch analysis, P^3 enables off-the-shelf analysers such as KLEE and AFL++ to be used unmodified as differential program analysers.
(2) A runtime system that enables such product programs to run as expected, with support for command-line arguments, global resources such as I/O streams, and program termination

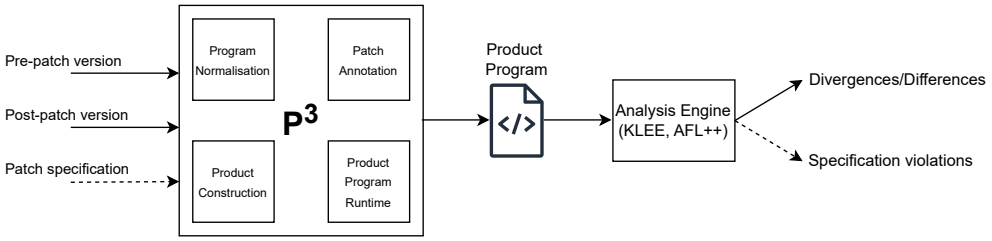

Fig. 1. End-to-end workflow for P³ (the dotted lines indicate optional elements).

(to avoid terminating the entire product program when only one of its constituents versions terminates).

(3) An experimental evaluation on complex patches from GNU Coreutils, together with a comparison against shadow symbolic execution [17], a comparable state-of-the-art technique that relies on *manually* merging programs. Our results show that P³ can find the same cross-version output differences as shadow symbolic execution when run in conjunction with KLEE, as well additional differences when run in conjunction with AFL++.

(4) An experimental evaluation on the patch specifications from our prior work [7], showing that P³ can find the same bugs as a previous approach where we manually constructed the product programs. We further extend the evaluation with additional patch specifications and show that we can apply P³ even without manually constructed test drivers.

The rest of the paper is structured as follows. §2 presents a motivating example, and §3 shows how P³ automatically handles various program constructs to create product programs. §3.1 discusses our program normalisation, §3.2 our program construction rules, §3.3 how patches are processed to provide the product program construction with information about which program parts have been added, removed, or modified, and §3.4 the runtime support provided by P³. §3.5 discusses implementation details, and §3.6 highlights the limitations. §4 evaluates our framework on a set of challenging patches and semantic patch specifications, and compares it with existing work that reasons about program paths via program merging and/or patch specifications. Finally, §5 discusses related work and §6 concludes.

2 Overview and Motivating Example

Figure 1 shows the end-to-end workflow for P³. Given a pre- and post-patch program together with an optional user-defined patch specification (e.g., as in our recent work [7]), P³ first processes the code to apply two transformations: program normalisation (§3.1), which rewrites both versions into a semantically equivalent form that reduces the number of different program constructs used (e.g., all loops are rewritten into `while` loops), and patch annotations (§3.3), which marks all the regions modified by the patch.

P³ then enters product program construction (§3.2), merging the two versions into a single one and injecting the necessary product program runtime support (§3.4) for dual execution. To detect control-flow divergences, P³ also inserts checks at every branch point in the program (§4.1). User-provided patch specifications are also processed accordingly (§4.2).

```c
int has_digit(const char *s) {
    int found = 0;
    while (*s != '\0') {
      if (*s >= '0' && *s <= '9')
        found = 1;
      s++;
    }
    return found;
}

int main(int argc, char **argv) {
  assert(argc == 2);
  if (has_digit(argv[1]))
    printf("Digits found\n");
  else printf("No digits found\n");
}
```

(a) Illustrative `has_digit` program.

```c
int has_digit(const char *s) {
    int found = 0;
    while (*s != '\0') {
      found ^= (*s >= '0' && *s <= '9');

      s++;
    }
    return found;
}

int main(int argc, char **argv) {
  assert(argc == 2);
  if (has_digit(argv[1]))
    printf("Digits found\n");
  else printf("No digits found\n");
}
```

(b) Buggy refactoring of `has_digit` program.

Fig. 2. Two versions of a simple program, used to illustrate the P^3 workflow.

The resulting product program can be fed directly into off-the-shelf analysis tools (e.g., KLEE or AFL++) to automatically discover inputs that trigger control-flow divergences, output differences or specification violations (§4).

We next show the envisioned end-to-end workflow of P^3 using a simple illustrative example. Shown in Figure 2, we consider a C utility `has_digit` that scans a NUL-terminated string to detect whether it contains digit characters. The original version, shown in Figure 2a, implements this functionality correctly: the variable `found` is set to 1 iff a digit character is found. The refactored version, shown in Figure 2b, aims to refactor the code by replacing the **`if`** statement with a more succinct update of `found`. However, instead of using the bitwise or (`|`) operator, which would have led to a correct refactoring, our hypothetical developer used the bitwise xor operator (`^`) by mistake. Therefore, the refactored version returns an incorrect result when the string contains an even number of digits (the cases with no digits or an odd number of digits are handled correctly).

Using P^3, the user can simply add a straightforward patch specification for this refactoring, in the form of a custom assert: `pp_assert(found1 == found2)`[1] just before the **`return`** statement. This assert expresses the user's intent that the refactoring should not have introduced any behavioural differences and the two versions of `has_digit` should behave identically.

As described above, P^3 produces a product program of the pre- and post-patch versions, following the various stages outlined in Figure 1. The resulting program produced by P^3 behaves as if executing both versions at once plus the aforementioned lightweight divergence and specification violation checks. It also handles the interaction with the environment, e.g, by duplicating the command-line arguments and giving each version its own set of file descriptors. Therefore, the resulting program can be analysed as is by off-the-shelf program analysers, which can be effortlessly used as differential program analysers. For instance, simply running the program produced by P^3 for the example in Figure 2 with KLEE or AFL++ immediately leads to the discovery of a patch specification violation and the generation of an input which triggers it.

The remainder of this paper describes in detail the various components of P^3 that allow it to deal with real-world patches and complex C programming constructs, and then showcases its strengths and limitations using a comprehensive evaluation.

[1] `pp_assert` is a custom function within the P^3 framework which correctly handles the state referred to in each version.

3 P³ Design

For the design of P³, we considered two important properties which together ensure that the result is useful for real-world programs:

(1) **Universality.** For any two given input programs, a single product program should be constructible, i.e. the construction is *universal*. Universality primarily impacts what C language constructs are transformed in which way.
(2) **Isolation.** When executed, the two component programs should be *isolated* from one another. For example, reading input in one component should not impact the other component's input stream. Isolation primarily impacts how the productised programs may interact with the environment and will be discussed in more detail in §3.4.

In the remainder of this section, we discuss how P³ builds product programs and what subset of the C language it covers. The construction is directly inspired by Eilers et al. [11], but extended to deal with the complexities of a real-world language like C, and with different program versions. One point to highlight is that P³ can handle the product construction of both entire programs, as demonstrated in §4.1, as well as partial systems, via the use of custom drivers, as shown in §4.2.

We start in §3.1 by describing the normalisations that are carried out as a pre-processing step to make the product construction more manageable. Then, in §3.2 we present the product program construction in detail, outlining how different program constructs are handled, including functions, branches and loops. §3.3 discusses how we feed patch information into the product construction process, and §3.4 discusses the P³ runtime. We end by discussing some implementation details in §3.5 and the main limitations in §3.6.

3.1 Program Normalisation

To reduce the number of different program constructs that need to be considered during product program construction, we start by normalising the program structure. For example, each `for` loop becomes an equivalent `while` loop.

Figures 3 and 4 show the most important normalisation rules in more detail. The actual prototype uses additional rules to give a more efficient result when additional constraints are satisfied (e.g., `do <body> while(<cond>);` ⇝ `{ <body> }` when `<cond>` evaluates to `0` without side effects and no `break` or `continue` statements appear directly in the loop) as well as rules to simplify the resulting code (e.g., `{{ <body> }}` ⇝ `{ <body> }`).

- **Loops.** All loops are normalised to `while` loops. The For and DoWhile rules remove `for` and `do ... while(...)` loops respectively.
- **Expression-level control flow.** All constructs that may perform control flow at the expression instead of the statement level—such as the comma operator (via Comma), short-cutting boolean logic (`&&` and `||` via LAnd and LOr), ternary conditional operators (`?:` via Ternary and Ternary'), and statement-expressions (`({/* ... */})`, a GNU extension, via StmtExpr and StmtExpr')—are hoisted to perform control flow at the statement level instead.
- **Function arguments and conditions.** Non-trivial (anything except for variable references and literals) function arguments, conditions, and arguments to `return` statements are replaced with temporary variables as per the FnArgs, While, If, Return and Return' rules. For example, `if (x != 0) { foo(x + 1); }` becomes `int t1 = (x != 0); if (t1) { int t2 = x + 1; foo(t2); }`.
- **Variable declarations.** All variable declarations are rewritten to only declare a single variable at a time (MonoDecl). Variable types, if applicable, are completed to contain explicit array bounds. For example, `int a[] = {0}, b;` becomes `int a[1] = {0}; int b;`. Thereafter,

For	`for(<init>; <cond>; <step>) <body>` ⇝ `{<init>; while(<cond'>) {{<body>} <step>;}}` `<cond'>` is `<cond>` if it exists and 1 otherwise.
DoWhile	`do <body> while(<cond>);` ⇝ `_Bool t = 1; while(t || (<cond>)) {t = 0; {<body>}}` `t` is a fresh identifier.
Comma	`<ctx>(<lhs>, <rhs>);` ⇝ `<lhs>; <ctx>(<rhs>);` `<ctx>(. . . )` represents the (potentially empty) surrounding expression.
LOr	`<ctx>(<lhs> || <rhs>);` ⇝ `_Bool t = (<lhs>); if(!t) {t = (<rhs>);} <ctx>(t);` `t` is a fresh identifier. `<ctx>(. . . )` represents the (potentially empty) surrounding expression.
LAnd	`<ctx>(<lhs> && <rhs>);` ⇝ `_Bool t = (<lhs>); if(t) {t = (<rhs>);} <ctx>(t);` `t` is a fresh identifier. `<ctx>(. . . )` represents the (potentially empty) surrounding expression.
Ternary	`<ctx>(<cond> ? <then> : <else>);` ⇝ `T t; if(<cond>) {t = (<then>);} else {t = (<else>);} <ctx>(t);` Applied if the type of `<cond> ? <then> : <else>` is a non-`void` type. `T` is the type of `<cond> ? <then> : <else>` and `t` is a fresh identifier. `<ctx>(. . . )` represents the (potentially empty) surrounding expression.
Ternary'	`<ctx>(<cond> ? <then> : <else>);` ⇝ `if(<cond>) {<then>;} else {<else>;} <ctx>(((void)0));` Applied if the type of `<cond> ? <then> : <else>` is a `void` type. `<ctx>(. . . )` represents the (potentially empty) surrounding expression.
StmtExpr	`<ctx>( ({<stmts> <expr>;}) );` ⇝ `T t; {<stmts> t = (<expr>);} <ctx>(t);` Applied if `<expr>` (and thus the statement expression) is of non-`void` type. Statement expressions are a GNU extension that execute a compound statement as an expression. `T` is the type of `({<stmts> <expr>;})` and `t` is a fresh identifier. `<ctx>(. . . )` represents the (potentially empty) surrounding expression.
StmtExpr'	`<ctx>( ({<stmts>}) );` ⇝ `{<stmts>} <ctx>( ((void)0) );` Applied if the statement expression is of `void` type. Statement expressions are a GNU extension that execute a compound statement as an expression. `<ctx>(. . . )` represents the (potentially empty) surrounding expression.

Fig. 3. Normalisation rules part 1. ⇝ denotes a single rewrite step. The rules are applied outside-in to all subterms until no rule matches (cascading normalisation to a fixed point).

function-level variable declarations with automatic storage duration that immediately initialise the variable are split into a declaration and a definition (rules SimpleInit, InitList and ArrayInit). For example, `int x = 0;` becomes `int x; x = 0;`.

- **Unstructured control-flow.** Unstructured control-flow, such as jumps using `goto` or `case` labels inside `switch` constructs are replaced with structured control flow, such as `if` statements. Our research prototype only implements this step in a limited fashion, which is enough to deal with many programs (see §3.6.1).

These normalisations are quite intricate in some cases, but also lengthy and their details do not contribute much to the discussion of this paper's topic. Nevertheless, the next section discusses one of the more interesting normalisations done, the variable declaration splitting that ensures that variable declarations are independent of any expressions that may have side effects.

3.1.1 Example Normalisation: Variable Declaration Splitting. The purpose of variable declaration splitting is to ensure that variables can be declared without causing any undefined behaviour or other side effects. This ensures that the product program construction can disable effects when they are not applicable (see, e.g., §3.2.4).

FnArgs	`<ctx>(<fn-arg>); ⇝ T t = (<fn-arg>); <ctx>(t);` Applied if `<fn-arg>` is used as a function argument and non-trivial. `T` is the type of `<fn-arg>` and `t` is a fresh identifier. `<ctx>(. . . )` represents the surrounding expression (including the function call and other arguments).
While	`while(<cond>) <body> ⇝ while(1) {if(!(<cond>)) {break;} {<body>}}` Only applied if `<cond>` is a non-trivial condition.
If	`if(<cond>) <then> [else <else>] ⇝ _Bool t = (<cond>); if(t) {<then>} [else {<else>}]` Applied if `<cond>` is a non-trivial condition. `t` is a fresh identifier.
Return	`return <expr>; ⇝ T t = (<expr>); return t;` Applied if `<expr>` is a non-trivial expression of non-`void` type. `T` is the type of `<expr>` and `t` is a fresh identifier.
Return'	`return <void-expr>; ⇝ <void-expr>; return;` Applied if `<void-expr>` is a non-trivial expression of `void` type.

MonoDecl	`<type> <var-1> [ = <init-1>], <var-2> [ = <init-2>] [, <more-vars>]; ⇝`
	`<type> <var-1> [ = <init-1>]; <type> <var-2> [ = <init-2>] [, <more-vars>];`
	Variable declaration syntax simplified. Actual rule applies to more syntactically complex declarations (e.g., `int x[] = {0}, (*y)[1] = &x; ⇝ int x[] = {0}; int (*y)[1] = &x;`).

SimpleInit	`<type> <var> = <expr>; ⇝ <type'> <var>; <var> = (<expr>);` Applied to non-`static` local variables when `<expr>` is an assignment expression (i.e., not a initialiser list). Note that `<type>` cannot be an array type as array types cannot be initialised by assignment. Variable declaration syntax simplified. Actual rule applies to more syntactically complex declarations. `<type'>` is `<type>` except that it is never `const`.

InitList	`<non-array-type> <var> = <init-list>; ⇝`
	`<non-array-type'> <var>; <var> = (char*)&(<non-array-type'>)<init-list>;`
	Applied to non-`static` local variables of non-array type when `<init-list>` is a initialiser list.
	Variable declaration syntax simplified. Actual rule applies to more syntactically complex declarations.
	`<non-array-type'>` is `<non-array-type>` except that it is never `const`.

ArrayInit	`<array-type> <var> = <init-list>; ⇝`
	`<array-type'> <var>; <size_t> i; char *p; p = (char*)&(<array-type'>)<init-list>;`
	`for(i = 0; i < <array-size>; ++i) { ((char*)&<var>)[i] = p[i]; }`
	Applied to non-`static` local variables of array type (thus `<init-list>` must be an initialiser list).
	Variable declaration syntax simplified. Actual rule applies to more syntactically complex declarations.
	`<array-type'>` is `<array-type>` except that it is never `const` and with explicit size.
	`<size_t>` is a type that is valid to represent array sizes on the target architecture (e.g., `unsigned long`).
	`i` and `p` are fresh identifiers.
	`<array-size>` is the size of an object of type `<array-type'>` in units `chars`.

While-Body	`while(<cond>) <body> ⇝ while(<cond>) {<body>}` Only applied if `<body>` is not a compound statement.
If-Body	`if(<cond>) <then> [else <else>] ⇝ if(<cond>) {<then>} [else {<else>}]` Only applied if `<then>` is not a compound statement or `<else>` exists and is not a compound statement.

Fig. 4. Normalisation rules part 2. ⇝ denotes a single rewrite step. The rules are applied outside-in to all subterms until no rule matches (cascading normalisation to a fixed point).

This normalisation applies to variable declarations with automatic storage duration, i.e., block-scope variables that are not **`static`** or **`extern`** (which implicitly also excludes `_Thread_local` variables), and actually have an initialiser to split off. Initialisation can happen in one of two main ways: Either a value is assigned (e.g., **`int`** `x = 0;`) or an initialiser list is used (e.g., **`int`** `x[] = {1};`).

For declarations that do not use an initialiser list, the splitting is straightforward (rule SimpleInit in Figure 4): The declaration is split into a declaration without the initialiser and an assignment of what was previously used to initialise the declaration. The only complication for this case comes from qualifiers, see the end of this section. Thus, **`int`** `x = 0;` becomes **`int`** `x; x = 0;`.

In the presence of an initialiser list, such as `int x = {0};` two more complications come up: First, initialiser lists are not expressions (and thus `int x; x = {0};` is not valid). Second, arrays can be initialised via an initialiser list, but not assigned.

The first issue is solved easily by transforming the initialiser list into a compound literal, which would, for example, transform `int x = {0};` into `int x; x = (int){0};`, This expression, which looks like an initialiser list that is cast to the target type, creates an unnamed variable that (due to this rewriting occurring at block level) has automatic storage duration. For anonymous types, the type needs to additionally be given a name (e.g., `struct { int a; } x = {0};` becomes `struct { int a; } S; struct S x; x = (S){0};`). See rule InitList in Figure 4.

While compound literals themselves function as expected for array types (i.e., `(int[]){1, 2}` is valid), the assignment needs to be replaced with a `memcpy`-style loop for declarations of variables whose outermost type is an array type (rule ArrayInit in Figure 4).[2] Thus, `int x[] = {1, 2};` becomes:

```
1 int x[2];
2 char* p;
3 p = (char*)&(int[2]){1, 2};
4 for (size_t i = 0; i < sizeof(int[2]); ++i) {
5   ((char*)&x)[i] = p[i];
6 }
```

Note that other normalisations, such as the previously mentioned loop normalisation will further transform this snippet in practice.

Type Qualifiers. The biggest limitation of this approach in general is that of `const`-qualified variables. Fundamentally, there is no way of splitting `const int x = 1;` into a declaration of `const int x;` and a later initialisation without invoking undefined behaviour due to ignoring the `const` rules. If only the outermost type is `const`-qualified, (e.g., `const int x = 0;` rather than `struct { const int a; } x = {0};`), our transformation removes that outermost `const`. While this is an observable change in theory (e.g., `_Generic(&x, int const*: 1, int*: 0)` changes depending on whether `x` is `const` or not), real and *valid* programs that do so are rare (`const` is used primarily to reject certain operations as invalid) and we did not encounter any in our evaluation.

The most common use of `const` in C programs is for the target type of pointers (e.g., `char const* s = "";`). Note that we do not strip these uses, nor do we need to, as the obvious splitting is perfectly valid: `char const* s; s = "";`. This includes more complicated types such as `char* const*`.

For this work we do not consider `volatile`-qualified variables, as not only are they only rarely used for purpose, but a product program of two versions of a program that requires explicit management of memory accesses, such as for DMA, is not all that useful.

3.2 Product Program Construction

Our automated product program construction leverages the modular product programs introduced by Eilers et al. [11]. The construction described in that work focuses on self-product programs for a toy language, and we extend the idea such that it works across multiple versions and for a complex real-world language like C. Since we make use of the normalisation step as described in the previous section, we only have to deal with a subset of program constructs.

Figures 5 and 6 give the formal rules of product program construction. Here, a product transformation $[\![(s_1, s_2)]\!]^{\langle p_1, p_2 \rangle}$ denotes the product of the corresponding statements s_1 from the pre-patch version and s_2 from the post-patch version, under activation variables (introduced in §3.2.1) p_1 and p_2. Vectors like $\vec{x}$ represent tuples of variables, expressions or types. In all rules, subscripts `1` and `2`

[2] This is not necessary if the array is a nested type, e.g., `struct { int a[1]; } S; struct S x = {0};` can be rewritten as `struct { int a[1]; } S; struct S x; x = (S){0};`

$[\![(\texttt{proc}\ f(\vec{u_1}:\vec{T_1})\ \texttt{returns}\ (r_1:R_1),$ $\texttt{proc}\ f(\vec{u_2}:\vec{T_2})\ \texttt{returns}\ (r_2:R_2))]\!]^{(p_1,p_2)}$	$= \texttt{proc}\ f_{product}(p_1, p_2, \vec{u_1}:\vec{T_1}, \vec{u_2}:\vec{T_2})$ $\texttt{returns}\ (r_1:R_1, r_2:R_2)$	(SIGN-PROD)
$[\![(\texttt{return}\ e_1,\ \texttt{return}\ e_2)]\!]^{(p_1,p_2)}$	$= \texttt{if}\ p_1\ \texttt{then}\ \{\, r_1 := e_1;$ $\texttt{if}\ \neg p_2^f\ \texttt{then return}\,(r_1,r_2);\ p_1^f = \texttt{false}\,\}$ $\texttt{if}\ p_2\ \texttt{then}\ \{\, r_2 := e_2;$ $\texttt{if}\ \neg p_1^f\ \texttt{then return}\,(r_1,r_2);\ p_2^f = \texttt{false}\,\}$	(RETURN-PROD)
$[\![(\texttt{return}\ e_1,\ \texttt{skip})]\!]^{(p_1,p_2)}$	$= \texttt{if}\ p_1\ \texttt{then}\ \{\, r_1 := e_1;$ $\texttt{if}\ \neg p_2^f\ \texttt{then return}\,(r_1,r_2);\ p_1^f = \texttt{false}\,\}$	(RETURN-PROD-1)
$[\![(\texttt{skip},\ \texttt{return}\ e_2)]\!]^{(p_1,p_2)}$	$= \texttt{if}\ p_2\ \texttt{then}\ \{\, r_2 := e_2;$ $\texttt{if}\ \neg p_1^f\ \texttt{then return}\,(r_1,r_2);\ p_2^f = \texttt{false}\,\}$	(RETURN-PROD-2)
$[\![(x_1 := \texttt{call}\ f(\vec{e_1}),\ x_2 := \texttt{call}\ f(\vec{e_2}))]\!]^{(p_1,p_2)}$	$= (t_1,t_2) := \texttt{call}\ f_{product}(p_1,p_2,\vec{e_1},\vec{e_2});$ $\texttt{if}\ p_1\ \texttt{then}\ x_1 := t_1;$ $\texttt{if}\ p_2\ \texttt{then}\ x_2 := t_2;$	(CALL-PROD)
$[\![(x_1 := \texttt{call}\ f(\vec{e_1}), \texttt{skip})]\!]^{(p_1,p_2)}$	$= \texttt{if}\ p_1\ \texttt{then}\ (x_1,\_) := \texttt{call}\ f(p_1, \texttt{false}, \vec{e_1}, \_);$	(CALL-PROD-1)
$[\![(\texttt{skip},\ x_2 := \texttt{call}\ f(\vec{e_2}))]\!]^{(p_1,p_2)}$	$= \texttt{if}\ p_2\ \texttt{then}\ (\_,x_2) := \texttt{call}\ f(\texttt{false}, p_2, \_, \vec{e_2});$	(CALL-PROD-2)
$[\![(\texttt{if}\ e_1\ \texttt{then}\ s_{t_1}\ \texttt{else}\ s_{f_1},$ $\texttt{if}\ e_2\ \texttt{then}\ s_{t_2}\ \texttt{else}\ s_{f_2})]\!]^{(p_1,p_2)}$	$= p_{t1} := p_1 \wedge e_1;\ p_{t2} := p_2 \wedge e_2;$ $p_{f1} := p_1 \wedge \neg e_1;\ p_{f2} := p_2 \wedge \neg e_2;$ $[\![(s_{t_1}, s_{t_2})]\!]^{(p_{t1},p_{t2})};\ [\![(s_{f_1}, s_{f_2})]\!]^{(p_{f1},p_{f2})}$	(IF-PROD)
$[\![(\texttt{if}\ e_1\ \texttt{then}\ s_{t_1}\ \texttt{else}\ s_{f_1}, \texttt{skip})]\!]^{(p_1,p_2)}$	$= p_{t1} := p_1 \wedge e_1;\ p_{f1} := p_1 \wedge \neg e_1;$ $[\![(s_{t_1}, \_)]\!]^{(p_{t1},\texttt{false})};\ [\![(s_{f_1}, \_)]\!]^{(p_{f1},\texttt{false})}$	(IF-PROD-1)
$[\![(\texttt{skip}, \texttt{if}\ e_2\ \texttt{then}\ s_{t_2}\ \texttt{else}\ s_{f_2})]\!]^{(p_1,p_2)}$	$= p_{t2} := p_2 \wedge e_2;\ p_{f2} := p_2 \wedge \neg e_2;$ $[\![(\_, s_{t_2})]\!]^{(\texttt{false},p_{t2})};\ [\![(\_, s_{f_2})]\!]^{(\texttt{false},p_{f2})}$	(IF-PROD-2)
$[\![(s_1,s_2);(t_1,t_2)]\!]^{(p_1,p_2)}$	$= [\![(s_1,s_2)]\!]^{(p_1,p_2)}\ ;\ [\![(t_1,t_2)]\!]^{(p_1,p_2)}$	(SEQ-PROD)
$[\![(x_1 := e_1, x_2 := e_2)]\!]^{(p_1,p_2)}$	$= \texttt{if}\ p_1\ \texttt{then}\ x_1 := e_1;\ \texttt{if}\ p_2\ \texttt{then}\ x_2 := e_2;$	(ASSIGN-PROD)
$[\![(x_1 := e_1,\ \texttt{skip})]\!]^{(p_1,p_2)}$	$= \texttt{if}\ p_1\ \texttt{then}\ x_1 := e_1;$	(ASSIGN-PROD-1)
$[\![(\texttt{skip},\ x_2 := e_2)]\!]^{(p_1,p_2)}$	$= \texttt{if}\ p_2\ \texttt{then}\ x_2 := e_2;$	(ASSIGN-PROD-2)

Fig. 5. Formalised product program construction rules for non-**while** constructs, assuming an already normalised program. Note that p_1^f and p_2^f in the RETURN-PROD rules refer to the function-level activation variables for the respective versions.

indicate pre- and post-patch statements, respectively, while in all associated code examples, we similarly use suffixes 1 and 2.[3]

In this section, we assume that for each statement s_1 in the pre-patch version, we have a corresponding statement s_2 in the post-patch version. For the case of a self-product program (where the two versions are identical, as considered in prior work [11]), this matching is of course trivial. In our work, we extend the matching to take code patches into account. Our extension is such that

[3]In our implementation, we use a prefix-based approach that is guaranteed to avoid name clashes, but in the paper we use this simpler naming convention for readability.

$[\![(\texttt{while } e_1 \texttt{ do } \{s_1\}, \texttt{while } e_2 \texttt{ do } \{s_2\})]\!]^{(p_1,p_2)}$ $= c_1 := p_1 \wedge e_1;\ c_2 := p_2 \wedge e_2;$ (WHILE-PROD)
$\texttt{while } c_1 \vee c_2 \texttt{ do}$
$\{\ cont_1 := \texttt{false};\ cont_2 := \texttt{false};$
$[\![(s_1, s_2)]\!]^{(c_1 \wedge \neg cont_1,\, c_2 \wedge \neg cont_2)};$
$\texttt{if } c_1 \texttt{ then } \{c_1 := p_1 \wedge e_1;\ \}$
$\texttt{if } c_2 \texttt{ then } \{c_2 := p_2 \wedge e_2;\ \}\}$

$[\![(\texttt{while } e_1 \texttt{ do } \{s_1\}, \texttt{skip})]\!]^{(p_1,p_2)}$ $= c_1 := p_1 \wedge e_1;\ \texttt{while } c_1 \texttt{ do}$ (WHILE-PROD-1)
$\{\ cont_1 := \texttt{false};\ [\![(s_1, \_)]\!]^{(c_1 \wedge \neg cont_1, \texttt{false})};$
$\texttt{if } c_1 \texttt{ then } \{c_1 := p_1 \wedge e_1\}\}$

$[\![(\texttt{skip}, \texttt{while } e_2 \texttt{ do } \{s_2\})]\!]^{(p_1,p_2)}$ $= c_2 := p_2 \wedge e_2;\ \texttt{while } c_2 \texttt{ do}$ (WHILE-PROD-2)
$\{\ cont_2 := \texttt{false};\ [\![(\_, s_2)]\!]^{(\texttt{false}, c_2 \wedge \neg cont_2)};$
$\texttt{if } c_2 \texttt{ then } \{c_2 := p_2 \wedge e_2\}\}$

$[\![(\texttt{break}, \texttt{break})]\!]^{(p_1,p_2)}$ $= c_1 := \texttt{false};\ c_2 := \texttt{false}$ (BREAK-PROD)

$[\![(\texttt{break}, \texttt{skip})]\!]^{(p_1,p_2)}$ $= c_1 := \texttt{false}$ (BREAK-PROD-1)

$[\![(\texttt{skip}, \texttt{break})]\!]^{(p_1,p_2)}$ $= c_2 := \texttt{false}$ (BREAK-PROD-2)

$[\![(\texttt{continue}, \texttt{continue})]\!]^{(p_1,p_2)}$ $= cont_1 := \texttt{true};\ cont_2 := \texttt{true}$ (CONTINUE-PROD)

$[\![(\texttt{continue}, \texttt{skip})]\!]^{(p_1,p_2)}$ $= cont_1 := \texttt{true}$ (CONTINUE-PROD-1)

$[\![(\texttt{skip}, \texttt{continue})]\!]^{(p_1,p_2)}$ $= cont_2 := \texttt{true}$ (CONTINUE-PROD-2)

Fig. 6. Formalised product program loop construction rules, assuming an already normalised program. Note that $c_1, c_2, cont_1$ and $cont_2$ in the rules for **`break`** and **`continue`** refer to the respective variables introduced in the surrounding **`while`** of the respective version. Any activation variables that depend on a changed variable need to take the change into account.

either the two statements continue to be of the same type (e.g., both `if` statements), or one of them is an empty statement (denoted in Figures 5 and 6 by `skip`). §3.3 discusses how the matching is done for the patched code.

3.2.1 Function Signatures. We start by adjusting the signature of the functions which are part of the product program being constructed. Figure 5 gives the formal SIGN-PROD rule and below, we present a concrete C-style example that instantiates it. Consider the function `f` with the following signatures in the two versions (the type of the return value and of the second parameter have changed across versions):

```
1 // version 1
2 ret_type_1 f(param_type1 x,
3              param_type2 y)
```

```
1 // version 2
2 ret_type_2 f(param_type1 x,
3              param_type3 y)
```

Then, its signature is changed to the following:

```
1 struct f_ret { ret_type_1 r1; ret_type_2 r2 };
2 struct f_ret f_product(bool p1, bool p2,
3                        param_type1 x1, param_type1 x2,
4                        param_type2 y1, param_type3 y2)
```

In particular, the following transformations are applied:

(1) The parameters, `x1`, `x2`, `y1` and `y2` represent the function parameters in the two versions: `x1` and `y1` for the first version, and `x2` and `y2` for the second version.
(2) The function returns a pair of values, the first with the type of the return value in the pre-patch version, and the second with the type of the return value in the post-patch version. This is accomplish by creating a new data type **`struct`** `f_ret` (each product function has a unique such **`struct`** associated with it).
(3) The parameters `p1` and `p2` are called *activation variables*. Variable `p1` controls whether the first version is active, while `p2` whether the second version is active. More exactly, these variables guard their version's statements, so that statements of a deactivated version are not executed. This allows the product program to deal with cases where the function might get called from only one of the versions: the function is called by setting the activation variable for that version to true, and the activation variable for the other version to false. The activation variables also allow the different versions to return at different times, as discussed below.

3.2.2 Function Returns. Note that returning from the product function is equivalent to returning from both component functions. Therefore, if one version reaches a **`return`** statement, it needs to wait until the other version also returns, if it is activated. This is accomplished by replacing a return statement with a pair of statements which update the corresponding entry in `ret`, and disable that version until the product program returns from that product function by setting the associated function-level activation variable to false. This is handled in Figure 5 by the RETURN-PROD rule (when a **`return`** is modified), RETURN-PROD-1 (when a **`return`** is removed), and RETURN-PROD-2 (when a **`return`** is added).

To illustrate these concepts, consider a function `f`, whose two versions are defined as follows:

```
// version 1
ret_type f(param_type x, param_type y) {
  return x;
}
```

```
// version 2
ret_type f(param_type x, param_type y) {
  return y;
}
```

Then, the product function `f_product` is:

```
struct f_ret { ret_type r1; ret_type r2 };
struct f_ret f_product(bool p1, bool p2,
                       param_type x1, param_type x2,
                       param_type y1, param_type y2) {
  struct f_ret ret;
  if (p1) {
    ret.r1 = x1;
    if (!p2) return ret;
    p1 = false;
  }
  if (p2) {
    ret.r2 = y2;
    if (!p1) return ret;
    p2 = false;
  }
}
```

3.2.3 Function Calls. Rules CALL-PROD, CALL-PROD-1 and CALL-PROD-2 in Figure 5 handle function call modifications, removals and additions, respectively. Consider the calls to the function `f` in the two versions:

```
// version 1
v = f(x, y);
```

```
// version 2
v = f(y, x);
```

The product program is as follows:

```
1 struct f_ret f_ret_val;
2 f_ret_val = f_product(p1, p2, x1, y2, y1, x2);
3 if (p1)
4   v1 = f_ret_val.r1;
5 if (p2)
6   v2 = f_ret_val.r2;
```

The product program construction is done such that the semantics of the two versions are maintained in the product program, while also having a single function call instead of duplicating the function calls from each version. The results are then assigned to the respective variables from both versions based on the activation variables, as shown in lines 3 to 6.

3.2.4 **`if`** *Statements.* The IF-PROD, IF-PROD-1 and IF-PROD-2 rules in Figure 5 deal with **`if`** statements. Recall that the normalisation stage has transformed all conditions to consist of only simple variables or literals, so they have no side effects, which is essential for the correctness of these rules.

To correctly handle **`if`** statements, we need to ensure that if a version's activation variable is false, neither condition nor body is executed. To see how this is handled, consider the following two versions:

```
1 // version 1
2 if (cond1)
3   then_expr1;
4 else
5   else_expr1;
```

```
1 // version 2
2 if (cond2)
3   then_expr2;
4 else
5   else_expr2;
```

The product program is as follows:

```
1 int p_t1 = p1 && cond1;
2 int p_t2 = p2 && cond2;
3 int p_f1 = p1 && !cond1;
4 int p_f2 = p2 && !cond2;
5
6 if (p_t1) then_expr1;
7 if (p_t2) then_expr2;
8 if (p_f1) else_expr1;
9 if (p_f2) else_expr2;
```

On lines 1-4, variables `p_t1`,`p_t2`,`p_f1`,`p_f2` store the four possible outcomes stemming from the two **`if`** conditions of the two versions, also taking into account whether the respective version is active. The `then` body (lines 6 to 7) and the **`else`** body (lines 8 to 9) are then guarded by these variables.

3.2.5 **`while`** *Loops.* After normalisation, the only loops remaining are **`while`** loops, which are handled by rules WHILE-PROD, WHILE-PROD-1, and WHILE-PROD-2 in Figure 6. Consider the following two versions:

```
1 // version 1
2 while (cond1)
3   // body1;
```

```
1 // version 2
2 while (cond2)
3   // body2;
```

Then, the product program is as follows:

```
1 int c1 = p1 && cond1;
2 int c2 = p2 && cond2;
3 while (c1 || c2) {
4   int cont1 = 0;
5   int cont2 = 0;
6   // product program of body1 and body2
7   if (c1) {
8     c1 = p1 && cond1;
9   }
10  if (c2) {
```

```
11      c2 = p2 && cond2;
12    }
13  }
```

The conditions of the **while** loops from both versions are stored in variables c1 and c2, also taking into account if the respective versions are active. This is done on lines 1 and 8 for the first version, and on lines 2 and 11 for the second version. The variables cont1 and cont2 are inserted to support **continue** statements, see §3.2.6.

The product program of body1 and body2 also uses these variables as activation variables. This allows the two versions to have different numbers of iterations. For instance, if one of the versions encounters a **break** statement, the respective activation variable is set to false, and the loop can proceed only for the other version, see §3.2.6.

3.2.6 **continue** *and* **break** *Statements.* The CONTINUE-PROD and BREAK-PROD rules from Figure 6 handle these constructs. We also illustrate the product construction rules with the following **while** statement with a **continue** inside:

```
1 // version 1
2 while (outer_cond1) {
3   if (inner_cond1)
4     continue;
5   // body1;
6 }
```

```
1 // version 2
2 while (outer_cond2) {
3   if (inner_cond2)
4     continue;
5   // body2;
6 }
```

For these versions, the product program is as follows:

```
1  int c1 = p1 && outer_cond1;
2  int c2 = p2 && outer_cond2;
3  while (c1 || c2) {
4    int cont1 = 0;
5    int cont2 = 0;
6    int c1_1 = c1 && !cont1 && inner_cond1;
7    int c1_2 = c2 && !cont2 && inner_cond2;
8    if (c1_1) {
9      cont1 = 1;
10   }
11   if (c1_2) {
12     cont2 = 1;
13   }
14   // product program of body1 and body2
15   if (c1)
16     c1 = p1 && outer_cond1;
17   if (c2)
18     c2 = p2 && outer_cond2;
19 }
```

As before, c1 and c2, defined in lines 1 and 2, store the conditions for the **while** loop for the two versions and act as activation variables for the body of these statements. We use additional flag variables, cont1 and cont2, defined on lines 4 and 5 to temporarily disable the remainder of an iteration for a version after a **continue** statement.

Since the **if** statements are nested inside the **while**, lines 6 and 7 define the *compound activation variables* c1_1 and c1_2 as a conjunction of the current activation variables, the corresponding **continue** flag variables, and the conditions from the **if** statement. These compound activation variables are then used to guard the body of the **if** statement of their respective versions. We note that deeply nested conditionals may, in theory, cause an exponential increase in code size under the product transformation. Such heavy nesting is typically uncommon given its implications on code quality and maintainability [24].

The **continue** statements are then handled by setting the respective flag variables. The statements in the loop body are hence no longer executed since they are guarded by *compound activation variables* that can only be true if `!cont1` or `!cont2`, respectively, is true. We also note that lines 16 and 18, which update the loop variables, are not guarded by their respective **continue** flags, as loop condition must be re-evaluated after a **continue** statement. In this way, the construction can handle cases where the two versions **continue** at different times, and thus have different numbers of iterations.

We handle **break** statements in a similar fashion, where we unset the respective loop activation variable (`c1` or `c2`) whenever a **break** is encountered, which not only prevents the body of the loop from being executed but also prevents the loop condition variables from being re-evaluated at the end of the loop, thereby allowing the respective version to exit the loop.

3.3 Patch Annotations

The product construction presented in §3 assumes that we have two aligned software versions as input, i.e. that we know which program constructs were added, removed or modified.

The *patch annotation* step presented in this section involves creating an annotated program that identifies the constructs added, removed or modified. It is applied immediately after the normalisation step in order to automate the subsequent product construction stage.

Our approach makes use of an abstract syntax tree (AST) matching algorithm. It works by comparing the ASTs of the pre- and post-patch versions of the code to identify changes. The algorithm proceeds as follows:

(1) **AST Construction:** Both the pre- and post-patch versions of the code are parsed to construct their respective ASTs. These trees represent the syntactical structure of the code, breaking it down into nodes that represent programming constructs like statements, expressions, and declarations.
(2) **Node Matching:** The algorithm attempts to match nodes from the pre-patch AST with nodes in the post-patch AST. This matching is done using a combination of structural similarity and node identifiers, considering the context and position of each node within the tree.
(3) **Change Detection:** Once the nodes are matched, the algorithm flags nodes that are added, modified, or deleted:
   - **Added Nodes:** Nodes present in the post-patch AST but not in the pre-patch AST.
   - **Modified Nodes:** Nodes that exist in both ASTs but differ in content (e.g., a change in an expression).
   - **Deleted Nodes:** Nodes present in the pre-patch AST but missing from the post-patch AST.

Consider the following illustrative example of a patch making a set of changes to the program:

```
1 // version 1
2 expr1;
3 if (cond1)
4   body1;
```

```
1 // version 2
2
3 if (cond2)
4   body2;
5 expr2;
```

To an AST matching algorithm (like the one used in CLANG-DIFF [21]), the aforementioned nodes are identified by making use of a tree traversal algorithm coupled with a set of heuristics. In the code above, lines 2 and 5 (highlighted red and green) show the identified deleted and added statements, respectively. Line 3 (orange highlight) shows the identified modified **if** condition between the two versions.

With these annotations in place, the product program construction can proceed as discussed in §3.2. The code below shows the product program for the example patch, highlighting the manner in which the previously identified added, deleted and modified nodes are dealt with:

```
1  if (p1) expr1;  // only in version 1
2
3  // Define the conditions based on the if statements
4  int t1 = p1 && cond1;  // Condition for the "then" branch of version 1
5  int t2 = p2 && cond2;  // Condition for the "then" branch of version 2
6  int f1 = p1 && !cond1;  // No "else" branch, so unused
7  int f2 = p2 && !cond2;  // No "else" branch, so unused
8
9  // Execute the "then" branches
10 if (t1) body1;
11 if (t2) body2;
12
13 if (p2) expr2;  // only in version 2
```

In particular, lines 1 and 13 show how the added and deleted statements in the product program are guarded by activation variables to ensure that the correct statements get executed for each version. Lines 4 to 7 show how the modified nodes as identified by our previous stage are dealt with in a product program, whereby the corresponding activation variables in the two versions use their respective condition expressions, `cond1` and `cond2`, for the first and second version respectively.

3.4 Product Program Runtime

Product programs cannot directly be executed in a normal program context, as their interaction with the outside environment is effectively duplicated. For example, their `main` function expects two sets of command-line arguments and returns two values. Beyond this simple issue related to program setup, *isolation* of the two component programs is an important property, as, for example, either version may try to use the standard input and output streams or attempt to terminate execution of the whole program.

To address these issues, we provide a runtime system which has three major tasks in the lifecycle of the product program:

(1) It starts the product program from a normal entry-point.
(2) It handles termination of the individual versions that make up the product program without terminating the whole product program.
(3) It manages global resources, such as access to standard I/O streams. This task is complicated in practice by having to deal with external calls, i.e. to functions whose code is unavailable.

Our presentation is done in the context of the C programming language (which our P³ prototype targets), but similar issues would also need to be addressed in other programming languages.

3.4.1 Startup. The `main` function of a product program has a form that is incompatible with that prescribed by the C programming language, as it duplicates the parameters and return value.

To start the program, the runtime provides an appropriate `main` function that initialises its global state and duplicates the command-line arguments and environment array before calling the product program's `main` function. This duplication is important, as the `argv` and `environ` arrays are writable in C, which means that each component version needs its own copy to prevent interference.

3.4.2 Shutdown. The runtime includes a standard library shim that is used to intercept calls to program termination functions such as `exit`. These shims disable the global activation variable for the currently active component version instead of terminating the whole product program. Shims for handler registration functions such as `atexit` register actions to be performed by our runtime instead of the standard library.

When the product program returns, the runtime gets the appropriate exit code from the return value or a global variable if one of the exit functions was called. It then computes the combined

return value as `(return1 == 0 ? 0 : 1) + (return2 == 0 ? 0 : 2)`, which makes it easy to determine whether each component version terminated normally.

3.4.3 Global Resources. The runtime manages access to global resources, such as environment variables (`envp`) and the standard I/O streams (`stdin`, `stdout`, and `stderr`) so that the component versions may read data without interfering with one another and write in an attributable manner. Since the standard I/O streams are directly accessible as global variables, and, even worse, are associated with well-known file descriptors (`0`, `1` and `2`), wrapping functions such as `printf` is not sufficient. Instead, the runtime will open a second set of I/O streams, making the two sets completely independent. For versions where such management is not really necessary (because the target program does not actually use standard I/O), this duplication can also be disabled.

To swap between the two sets of I/O streams, the product program is instrumented with calls to runtime functions `libpp_activate1()` and `libpp_activate2()` which will enable the corresponding streams.[4] To assist with debugging, calling `libpp_deactivate()` returns the program to a neutral state, in which any usage of these shared resources is an error. This way, it is possible to not only provide the product program with identical input for its component versions, but also to allow for intentional differences due to changes in the interface.

Environment variables are handled mostly the same with the additional complication that the three-value form of `main` also requires a pointer to the appropriate set of environment variables to be passed.

3.4.4 External Calls. Source code may not be available for all functions that are called by the target program. P^3 provides multiple ways of dealing with this issue.

First, a handwritten product function can be provided; this is similar to having environment models in program analysis techniques such as symbolic execution [6]. For common library functions, such handwritten models can provide effective solutions that can be widely reused.

Alternatively, a normal non-product function can be called for each version as-is, as long as it is stateless or idempotent. To do so, we first call `libpp_activate1()` or `libpp_activate2()` to enable the appropriate global state for the standard library, before calling the function with appropriate version-specific arguments. Afterwards, we call `libpp_deactivate()` to return to a neutral state (see also §3.4.3).

Finally, if a product function cannot be provided and the function is (possibly indirectly) stateful, we support a mechanism to duplicate that state by duplicating and renaming the binary object file, so that two versions of the function are created that do not interfere with one another. We show the viability of this approach by using it for `getopt`, which uses global variables to keep the parsing state in between calls. The object file is duplicated and each symbol is changed to have a unique version-specific name.[5]

3.5 Implementation

Our prototype implementation P^3 begins by normalising both input versions of a given C program at the source code level, as per §3.1. This normalisation step also resolves macros and **`#include`** statements. In addition to reducing the number of language constructs that the product program construction has to consider, this has the added benefit of improving the semantic matching when generating patch annotations (§3.3). The normaliser uses Clang Libtooling [8] to repeatedly parse and simplify the source code.

[4]This also changes `errno` and `environ` to that of the respective component version.

[5]To support the experiments discussed in §4, we do not actually perform this operation at the binary level, but rather on LLVM bitcode files, as this is the level that KLEE operates on. When this is not necessary, the renaming can also be performed at the binary level, which we validated using a combination of `nm` and `objcopy` driven by a small Python script.

For the patch annotations discussed in §3.3, we follow a scheme similar to the `change()` patch annotations presented in SHADOW [28]. However, unlike SHADOW our technique is *fully automated* and relies on an AST matching algorithm for identifying added, deleted and modified nodes between the ASTs of the pre- and the post-patch versions. In our implementation, we leverage the algorithm of CLANG-DIFF [21]. Note that even when this heuristic step yields suboptimal results, our product program construction still yields a correct product program.

Next, we perform the actual product program construction, which is also implemented using CLANG LIBTOOLING. To do so, we apply the transformations described in §3.2, where function signatures, function calls, **`if`** statements, and **`while`** loops are transformed to accommodate the product semantics. This is also the stage where the necessary patch specifications are handled. Moreover, since we expect a developer to write a patch specification while writing the patch itself, we make use of a custom `pp_assert` statement which then gets picked up at this product construction stage and transformed into source-level assertions which are compatible with the off-the-shelf analysers.

3.6 Limitations

For our research prototype, we have focused on universality (cf §3) and only implemented isolation to the point of finding general strategies and until our prototype was sufficient for our evaluation. In the two following sections we will look closer at these two properties.

3.6.1 Universality: Normalisation and Product Construction. While the P³ prototype described tries to handle most of the cases as encountered within our representative, real-world benchmarks, we acknowledge that not all C constructs are covered. In particular, our research prototype is unable to perform normalisation for programs using arbitrary **`goto`**s or complex **`switch`**/**`case`** constructs such as Duff's device. We acknowledge the fact that our prototype is not fully universal and would require further work for full coverage. However, given the richness and flexibility for a language like C, it is not uncommon for such source-to-source prototypes to exhibit certain limitations, as outlined by e.g., the popular CIL infrastructure [26].

Moreover, for the specific case of unrestricted **`goto`**s whose use has long been deemed harmful [10], we have only run into a single instance of a backwards **`goto`** in the `cp` benchmark. Instead of discarding this benchmark, we decided to manually refactor the code to remove the construct and continued with the evaluation of the patch which then gave us the expected outcome.

3.6.2 Isolation: Runtime and Environment Handling. The global state managed by `libpp_activate1()` and `libpp_activate2()` (see §3.4.3) does not handle all of the global state exposed by the standard library, although some commonly used functionality is covered. Probably the most important unisolated resource is the file system, which we omitted as in our benchmarks there are no file writing conflicts between the two versions. We note that this is a common limitation shared with other program analysers such as most fuzzers.

To achieve the strongest isolation possible, one would need to intercept file interactions (either via a source-based approach as we did for input/output streams, or via a lower-level solution such as intercepting system calls), and perform the results in two isolated namespaces. These can be created in various ways, ranging from Linux namespaces, as used by containerisation software, to some specialised manipulation of paths when opening files. Completely modelling the environment is of course also possible.

On the pure implementation side, our prototype runtime is not thread-safe. This is not a fundamental issue, as the usage of `libpp_activate1()`, `libpp_activate2()` and `libpp_deactivate()` (see §3.4.3) effectively forms critical sections, access to which could easily be synchronised.

4 Evaluation

We evaluate our framework P^3 on the collection of GNU Coreutils [13] patches from CoREBench [5]. We conduct two distinct experiments, evaluating the different ways in which we envision P^3 to be useful for testing software patches.

First, in §4.1 we perform *whole-program-based validation*, where a product program (of the pre- and post-patch versions) of the entire system is constructed and analysed for control-flow divergences and output differences, in a manner similar to Shadow [17, 28].

Second, in §4.2 we explore *patch specification-based validation* (based on the idea of patch specifications from our earlier work [7]) and explore the challenges faced in writing meaningful patch specifications and their efficacy in exposing potential bugs when used in conjunction with our P^3 framework.

Our evaluation is designed to assess three strengths of P^3: (1) its ability to automatically construct product programs for real-world code (C in our prototype implementation), (2) its interoperability with different program analysers (in particular AFL++ and KLEE in our evaluation); and (3) its ability to deal with patch specifications and both modular and whole-program analysis.

Note that the key strength of P^3 is to make arbitrary program analysers be differential program analysers. Our aim is not to directly compare with the many different specialised techniques for differential program analysis, but rather show that *off-the-shelf* analysers can be effectively used for differential program analysis without requiring any modifications.

4.1 Whole-Program-Based Validation

In this section, we explore the use case of whole-program-based validation using our P^3 framework. The core idea here is that by using our P^3 framework and automatically constructing a product program that combines both the pre- and post-patch versions of the entire system, we can use multiple off-the-shelf analysers to identify control-flow divergences (where the two versions take different branches at a conditional statement) and output differences (where the two versions produce different outputs).

This evaluation methodology is inspired by the Shadow project [17, 28]. Shadow proposes to merge the pre- and post-patch versions using an approach based on manual annotations, and then uses a new form of symbolic execution (called shadow symbolic execution) to detect control-flow divergences and output differences. Such divergences and differences can be valuable to developers who can check whether they are intended changes or unintentional bugs.

Compared to Shadow, we merge programs in a fully automatic fashion and are not constrained to using symbolic execution, but can employ different off-the-shelf techniques. To demonstrate this latter advantage, we use our P^3 framework with both KLEE (denoted as P^3_{KLEE}) and AFL++ (denoted as $P^3_{\text{AFL++}}$) and in the process demonstrate their complementary advantages.

For our whole-program-based validation, we try to use the same experimental setup as in Shadow [17, 28]. Shadow starts with test cases that reach the patch, and then explores post-patch paths to discover those resulting in control-flow divergences and/or output differences.

We use the same methodology and the same suite of patches from CoREBench in our evaluation. CoREBench [5] is a suite of complex real-world patches and bugs that is often used in the evaluation of patch testing techniques. The *bug-inducing commits* have been given an ID within CoREBench. In particular where the IDs appear as 5=16 and 12=17, it denotes that the commit was exactly the same but resulted in multiple bugs getting introduced.

For each *bug-inducing* CoREBench patch, we automatically construct a product program. This product program is then analysed by both KLEE and AFL++, representative of two popular program analysis techniques, dynamic symbolic execution and fuzzing, respectively.

Table 1. Experimental results for P^3_{AFL++} and P^3_{KLEE} in comparison to Shadow. The results for P^3_{KLEE} and Shadow are identical. Cells highlighted in grey represent cases where P^3_{AFL++} performs worse (light grey) or better (dark grey) than Shadow and P^3_{KLEE}. **Div.** column is short for **Divergences** and indicates whether the approach was able to find control-flow divergences between the two versions.

ID	Shadow			P^3_{KLEE}			P^3_{AFL++}		
	Div.	Output diff.		Div.	Output diff.		Div.	Output diff.	
		Expected	Bug		Expected	Bug		Expected	Bug
1 (rm)	✓	No	No	✓	No	No	✗	No	No
3 (cut)	✓	No	No	✓	No	No	✗	No	No
4 (tail)	✓	Yes	No	✓	Yes	No	✗	No	No
5=16 (tail)	✓	No	No	✓	No	No	✗	No	No
6 (cut)	✓	No	Yes	✓	No	Yes	✗	No	No
7 (seq)	✓	No	No	✓	No	No	✓	No	Yes
8 (seq)	✓	No	No	✓	No	No	✓	No	Yes
10 (cp)	✓	No	No	✓	No	No	✗	No	No
11 (cut)	✓	No	No	✓	No	No	✗	No	No
12=17 (cut)	✓	No	Yes	✓	No	Yes	✗	No	No
13 (ls)	✓	No	No	✓	No	No	✗	No	No
14 (ls)	✗	No	No	✗	No	No	✗	No	No
15 (du)	✗	No	No	✗	No	No	✓	Yes	No
19 (seq)	✓	No	No	✓	No	No	✓	No	Yes
21 (cut)	✓	No	Yes	✓	No	Yes	✗	No	No
22 (expr)	✗	No	No	✗	No	No	✗	No	No

At each branch of the product program, we add instrumentation to check for control-flow divergences. This is a simple check that introduces an `if` statement comparing the compound activation variables of the two versions at that branch point. If a divergence is detected, the program is crashed (with a certain exit code, to distinguish it from other types of crashes), which will be detected by the underlying analysers.

P^3_{KLEE}. The methodology followed while running product programs with KLEE roughly follows Shadow, which is also based on KLEE. We take one input from the regression test suite as an initial seed for KLEE, running in concolic mode. Then, starting from the states reaching control-flow divergences, we continue exploring paths in breadth-first mode. We run KLEE for 10 min and save all the generated inputs. We then run all these inputs to find those resulting in output differences or bugs. Concretely, we do so by running the inputs on the pre- and post-patch versions of the utility, and comparing the outputs. We perform these runs under Valgrind [27], to find memory errors.

P^3_{AFL++}. For AFL++, we run the fuzzer for 10 min on each constructed product program, with inputs from the regression test suite of the particular utility that reach the patch being used as initial seeds. During the fuzzing process, we collect all the inputs leading to crashes caused by control-flow divergences. As for the KLEE-generated inputs, these inputs are then replayed on the pre- and post-patch versions of the utility (with Valgrind enabled) to identify any cases that result in output differences or bugs.

Table 1 summarises the results obtained while following the aforementioned methodology on the bug-inducing patches from CoREBench, along with a comparison with Shadow.

Table 2. Bugs and output differences identified by the P^3 framework which were missed by SHADOW.

ID	Input	Pre-patch output	Post-patch output	Classification

7 (seq)	`0 3 1`	`0`	`0`
			`1`
			`2`
			`3`
				Bug

8 (seq)	`-0`	**No output**	`1`
			`2`
			`3`
			`4`
			`5`
			`6`
			`7`
			`8`
			`9`
			`10`
				Bug

15 (du)	`-x / dir`	`4 /opt`
		`0 /dev`
		`8 /root`
		`0 /sys`
			`4 /opt`
			`8 /root`
				Expected

19 (seq)	`-w -2 .81`	`-2`
		`-1`
		`0`
			`-2`
			`-1`
			`00`
			`01`
				Bug

One of the key highlights is that P^3_{KLEE} finds all the divergences and bugs found by SHADOW. Importantly, this is done via an automated product program construction, while the merged programs in SHADOW require manual annotations.

The other key highlight is that P^3_{AFL++} offers complementary strengths compared to P^3_{KLEE}. This shows the advantage of being able to run different program analysers in P^3. In particular, P^3_{AFL++} misses the bugs introduced by three of the patches in `cut`, as well as an output difference in one of the `tail` patches, which are found by P^3_{KLEE}. However, it finds new bugs and output differences compared to SHADOW and P^3_{KLEE}: it finds the bugs introduced by the three patches in `seq`, as well as an output difference introduced by the `du` patch. These are highlighted in Table 2, which shows concrete inputs generated by P^3_{AFL++}, together with the output of the pre- and post-patch versions.

Case Study: Regression Bug in `seq`. The utility `seq` is used to print sequences of numbers. For instance, `seq 10 2 18` prints the numbers from `10` to `18`, with a step of `2`. That is, it prints:

```
10
12
14
16
18
```

In GNU COREUTILS-8.20, an optimisation was added to `seq` to simplify the common case of counting with a step of `1` over non-negative integers when no custom formatting is used. The

Table 3. Patch specifications from [7]. Only the critical assertion is shown. The variables in the two versions are denoted by prefixes `_PP1_` and `_PP2_`.

ID	Patch specification	Explanation
14 (ls)	`0 == strcmp(_PP1_p, _PP2_p)`	Refactoring patch, thus the output of the function should remain unchanged.
22 (expr)	`_PP2_v->u.i == _PP1_v->u.i`	Introduction of a new error case and partial refactoring that should not lead to **`struct`** `v` changing.
8b (seq)	`_PP2_buf == _PP2_z || (_PP2_z[-1] >= '0' && _PP2_z[-1] <= '9')`	Modifies output logic of `seq` such that the output buffer now exactly mirrors the temporary `z` variable.

intended aim of this patch was to detect when all operands are plain decimal integers, the requested step is `1`, and no format string is given, and then invoke an internal function `seq_fast` that increments the string representation directly:[6]

```
1 if (format_str == NULL
2     && all_digits_p (argv[1])
3     && (n_args == 1 || all_digits_p (argv[2]))
4     && (n_args < 3 || STREQ ("1", argv[ 3 ]))) {
5   // ...
6   seq_fast(s1, s2);
7   exit(EXIT_SUCCESS);
8 }
```

Unfortunately, this optimisation was incorrect: due to an indexing mistake (as highlighted by the grey box in the listing above), the patched code tested the *end* operand (which is the third argument) in `seq` rather than the *step* operand (which is the second argument). Despite the optimisation being incorrect, insufficient testing meant that it made it into an official release, only for the bug to be discovered and reported later.[7]

The patch touched a single file, but altered 148 lines of code across 5 different code regions (hunks). Our P³ framework automatically constructs a product program that runs together the correct, pre-patch and the buggy, post-patch version, demonstrating that it can handle large complex real-world patches.

Control-flow divergence checks, as described earlier in the section, are automatically inserted by P³ throughout the product program. For each input triggering control-flow divergences across versions, the outputs are compared and differences flagged. With these checks in place, off-the-shelf AFL++, running the product program produced by P³ was able to generate input `0 3 1` within 10 seconds, which triggered the bug. More precisely, run with this input, the correct pre-patch implementation prints `0`, but the incorrectly optimised post-patch version incorrectly takes the fast path (because the end value and not the step value is compared to `1`) and prints the numbers `0, 1, 2, 3` (one per line).

4.2 Validating Patch Specifications

In order to show the effectiveness of the P³ framework when used in conjunction with patch specifications, we carry out two related experiments that use (partial) patch specifications to find multiple real-world bugs from the CoREBench suite.

[6] https://cgit.git.savannah.gnu.org/cgit/coreutils.git/commit/?id=77f89d014be68e42de5107aee0be95d18ee1735c
[7] https://debbugs.gnu.org/cgi/bugreport.cgi?bug=13525

Table 4. Summary of the results obtained by our P^3 framework on the patches from [7] with KLEE and AFL++.

ID	Kind	LoC	KLEE			AFL++	
			Time	Paths	Violation	Time	Violation
14 (ls)	Inducing	306	2 s	134	Found	1 s	Found
	Fixing	313	2 s	380	Found	1 s	Found
22 (expr)	Inducing	1827	1 s	6	Found	1 s	Found
	Fixing	1827	3 s	11	None	10 min	Not found
8b (seq)	Semantic	750	14 s	1121	None	10 min	Not found

The first experiment uses the patch specifications from [7] along with their test drivers, but using P^3 to automatically construct product programs. By demonstrating that these specifications and their drivers can be used with P^3, we aim to establish that P^3 can automate patch testing using naturally expressed patch specifications.

For the second experiment, we add four more examples from CoREBench and write our own patch specifications. Unlike for the first experiment, we do not use custom test drivers for these patch specifications. With this experimental setup we want to explore how well P^3 works when running full-fledged real-world programs with patch specifications.

4.2.1 Patch Specifications with Custom Test Drivers. In our earlier work [7], we wrote patch specifications for three different patches in three different GNU Coreutils utilities (`ls`, `expr` and `seq`), together with custom test drivers that call the function modified by the patch. A short description of these patch specifications can be found in Table 3.

We extract the functions that are modified by the patch, add the patch specifications and then compute their product program using P^3. To run the resulting product function, we slightly modify the driver so that it can be called as a product `main` function. The LoC column in Table 3 refers to the number of lines of code in the product programs automatically constructed by P^3 for those functions and test drivers. Table 4 summarises our results, which we discuss in more detail below.

`ls` refactoring validation. CoREBench issue #14 contains a patch that was intended to refactor `ls`. We first use P^3 for the *bug-inducing commit* whereby we construct a product program of the original and the refactored code and run it under P^3_{KLEE} and $P^3_{\text{AFL++}}$. Both tools are able to find an input that violates the patch specification. We manually verified that this indeed induces the expected bug.

We then create a compound patch by combining the buggy patch with its supposed fix, also taken from CoREBench. For this second patch, we also use P^3 to construct the product program followed by an analysis step with P^3_{KLEE} and $P^3_{\text{AFL++}}$. Our findings match the ones from our earlier work [7], as we notice another violation of the specification due to a (probably benign) difference in path concatenation. Providing both full versions of `ls` with input that causes a violation of the patch specification shows no visible difference, which is in line with our previous findings [7]. Both aforementioned specification violating inputs were found almost instantaneously with either tool. Similar results can be observed when using manually constructed product programs [7].

`expr` partial refactoring validation. We follow the same strategy for this example, by starting with the bug-inducing commit for CoREBench issue #22. Again, both P^3_{KLEE} and $P^3_{\text{AFL++}}$ quickly find a violation of the patch specification, which we manually verified to induce the expected bug.

Table 5. Additional patch specifications attempting to capture developer intent. The variables in the two versions are denoted by prefixes `_PP1_` and `_PP2_`.

ID	Patch specification	Explanation
3 (cut)	`_PP2_n_rp == _PP1_n_rp - 1`	The post-patch `n_rp` value should be one less than the pre-patch value after the addition of logic that merges finite range pairs.
7 (seq)	`_PP2_argv[1] == _PP1_first.value && _PP2_argv[2] == _PP1_last.value`	Since this is an optimisation patch for a special case, the new start and end values of a range should be the same as when the optimisation was not being used.
8 (seq)	`_PP2_z == _PP2_buf || *(_PP2_z - 1) == '\n'`	Ensures that after the first number is written, a newline is not incorrectly added unless explicitly intended by the separator. This maintains the correct sequence format when using non-newline separators, aligning with the commit's intent to allow flexible separators.
12=17 (cut)	`*_PP1_range_start == *_PP2_range_start`	The patch adds an option to `cut` but the role of `range_start` within `print_kth` function should remain unaffected by this new complementary option.

Unlike in the previous case, the patch specification holds for the fixed patch, which incidentally makes validation slightly harder, as there is no specification-violating input to check. In addition to manually checking some arbitrarily chosen inputs, we also note that the number of paths generated by the bounded, but otherwise complete symbolic exploration of the product program is exactly the same for both our product program and for the program from [7]. Despite the number of paths being the same, we observe an expected increase in the total number of executed instructions, as the automated product program construction is more verbose than the manually crafted one. As AFL++ will never terminate on programs where it does not detect a bug, we used a time budget of 10 min, which it reached without finding any specification violation.

seq semantic change validation. For our final case study, we validate the bug-fixing patch for `seq` from [7] as a semantic change, rather than by using it to correct an earlier mistake. As in [7], we use the buggy version of `seq` as the base version and have the patch specification describe the intended change. Again, we use both P^3_{KLEE} and $P^3_{\text{AFL++}}$ to construct the product program and to look for any violations. As expected, there are no violations.

The number of paths explored by P^3_{KLEE} is the same as in the artifact for [7], while the number of explored instructions again increases. As before, we run $P^3_{\text{AFL++}}$ for 10 min during which no violations of our patch specifications were detected.

4.2.2 Patch Specifications without Custom Test Drivers. The setup for this set of experiments aims to demonstrate the efficacy of our P^3 framework when used in conjunction with patch specifications on full real-world programs, i.e. without the use of any custom test drivers. We believe that this setup provides a lightweight method of robustly testing patches without the manual overhead of continuously writing test drivers.

Table 5 shows the critical part of the patch specifications we wrote, along with a rationale for what property they validate. Understanding the intended semantic change (or lack thereof in the case of refactorings) is critical not only for writing these patch specifications, but also for reviewing patches in general. In fact, our ability to deduce what we believe to have been the original developers' intent played a central role in the selection of patches for this experiment. While we tried to write patch specifications that could reasonably have been written to support the bug-inducing patch, we of course benefited from some hindsight while writing them.

We used P^3 to generate two product programs for each of the chosen benchmarks: one over the original and the buggy version and one over the original and the fixed version. This enables us to validate that the patch specification is strict enough to catch the bug and allows the fixed patch to pass. We used test cases extracted from the bug reports to manually validate that this is indeed the case. After this initial validation step, we make use of P^3's ability to work with a variety of testing techniques by analysing these product programs with patch specifications with both AFL++ and KLEE. For all four cases, both $P^3_{\text{AFL++}}$ and P^3_{KLEE} were able to find specification violations for the buggy version of the patches. No specification violations were found for the fixed versions of the patches (within a timeout of 10 min for both KLEE and AFL++).

5 Related Work

The automated product program construction described in this paper is built on the modular product programs presented by Eilers et al. [11], whose ideas we extend to cover much of the C programming language and handle *different* program versions. Product programs have been originally presented as a form of self-composition to reduce an information-flow hyper-property relating two program traces to a safety property in the composed program [3]. A concrete method for computing the composition in a simple language was given in [2].

A large number of automated techniques for validating patches have been proposed in the literature [9, 14, 15, 17, 23, 25, 32–35]. These techniques rely on specialised program analyses, while our approach can use *off-the-shelf* unmodified analysers once the product programs are constructed. Furthermore, these techniques suffer from the oracle problem; P^3 makes it easy to write patch specifications [7], which we explore in this work.

Product programs have been used in differential assertion checking [19] to determine whether errors are introduced by a new program version. While the framework could in principle handle the kind of patch specifications we envision, it focuses exclusively on generic errors such as buffer overflows. Reasoning is done using static analysis via Boogie [1], which imposes different requirements on the construction of product programs.

Similarly, product programs have been proposed for relational verification of programs [2], by way of reducing relational Hoare logic quadruples to standard Hoare triples, which can then be verified using standard techniques. Our work expands on that approach by computing product programs for general C programs with only minor restrictions (e.g., no unrestricted `goto` usage), and shows that it is not restricted to (relational) Hoare logic or more generally to verification.

Product programs have also been used to find issues with conflicting patches by automatically verifying semantic conflict-freedom, a property that describes the disjointedness of two patches versus a single base version on a semantic level, rather than a purely textual or syntactic one [31].

Yi et al. [36] propose the notion of *change contracts* for evolving software, which are similar in spirit to patch specifications, but based on the use of a specification language. While both P^3 and change contracts support cross-version reasoning, they differ in their approach: P^3 reduces the problem to standard program analysis by transforming version pairs into product programs amenable to off-the-shelf analysers, thereby turning any off-shelf-analysers into differential program

testers, while change contracts introduce a dedicated specification language and bespoke verification engine to express and check intended behavioural changes.

Other forms of multi-version programs have been proposed in the past, e.g., in work on shadow symbolic execution [17], multi-version interprocedural control flow graphs [20], regression model checking [35], and GPGPU verification [4].

Over the past decade, automated generate-and-validate program repair techniques have garnered significant attention. These techniques tend to generate many overfitted patches [22, 30] that need to be rejected in the validate step. In addition to the usage of pre-existing test suites and implicit oracles (e.g., checking if compilation succeeds), various heuristic solutions have been proposed to reject plausible but incorrect patches during validation [9, 14, 15, 25, 32–34]. Our proposed approach can reject patches that have subtle semantic errors without encountering false positives, but requires (presumably human-written) patch specifications to do so, thus being unsuited for quick rejection of large numbers of automatically generated patches. Once a repair attempt has been decided upon, our framework enables the developer to reason about whether the automatically generated patch exactly achieves a stated goal, helping a human developer validate their understanding of machine-generated code.

6 Conclusion

Motivated by the observation that it is often easier to describe the differential behaviour of a patch than to fully specify the intended behaviour of a program, we present P³, a framework for automated reasoning about patches via product programs. In this paper, we have shown that product programs can be computed automatically even for large and non-trivial C programs and patches, and that the resulting product program can be used by off-the-shelf tools to validate patch behaviour. We have also demonstrated that partial patch specifications can be easy to formulate and still catch many practically relevant bugs by encoding assumptions about how the system is supposed to change.

Through the use of our P³ framework, we demonstrate the advantages of an analysis-agnostic framework, where we found analysis techniques as varied as AFL++ and KLEE to be complementary when used in conjunction with our framework on the challenging set of patches from the CoREBench suite.

7 Data-Availability Statement

Our artifact is available on Zenodo [29], with the latest version accessible at https://srg.doc.ic.ac.uk/projects/p3/.

Acknowledgements

We would like to thank Alastair Donaldson and Martin Nowack for feedback on the text, and Manuel Carrasco for feedback on the artifact. This project has received funding from the European Research Council under the European Union's Horizon 2020 research and innovation program (grant agreement 819141).

References

[1] Michael Barnett and K. Rustan M. Leino. 2005. Weakest-Precondition of Unstructured Programs. In *Proc. of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'05)* (Lisbon, Portugal). 82–87. https://doi.org/10.1145/1108768.1108813

[2] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *Proc. of the 17th International Symposium on Formal Methods (FM'11)* (Limerick, Ireland). https://doi.org/10.1007/978-3-642-21437-0_17

[3] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011), 1207–1252. https://doi.org/10.1017/S0960129511000193

[4] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. 2012. GPUVerify: A Verifier for GPU Kernels. In *Proc. of the 27th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'12)* (Tucson, AZ, USA). https://doi.org/10.1145/2384616.2384625

[5] Marcel Böhme and Abhik Roychoudhury. 2014. CoREBench: Studying complexity of regression errors". In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'14)* (San Jose, CA, USA). https://doi.org/10.1145/2610384.2628058

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08)* (San Diego, CA, USA).

[7] Cristian Cadar, Daniel Schemmel, and Arindam Sharma. 2023. Patch Specifications via Product Programs. In *Proc. of the 11th International Conference on Formal Methods in Software Engineering (FormaliSE'23)* (Melbourne, Australia). https://doi.org/10.1109/FormaliSE58978.2023.00012

[8] Clang LibTooling 2023. LibTooling. https://clang.llvm.org/docs/LibTooling.html.

[9] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*. 18–25. https://doi.org/10.1109/IBF50092.2020.9034714

[10] Edsger W Dijkstra. 1968. Letters to the Editor: Go To Statement Considered Harmful. *Communications of the Association for Computing Machinery (CACM)* 11, 3 (1968), 147–148.

[11] Marco Eilers, Peter Müller, and Samuel Hitz. 2020. Modular Product Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 1 (2020). https://doi.org/10.1145/3324783

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

[13] Free Software Foundation, Inc. 2025. GNU Coreutils. https://www.gnu.org/software/coreutils/.

[14] Charaka Geethal, Marcel Böhme, and Van-Thuan Pham. 2023. Human-in-the-Loop Automatic Program Repair. *IEEE Transactions on Software Engineering (TSE)* 49, 10 (2023), 4526–4549. https://doi.org/10.1109/TSE.2023.3305052

[15] Ali Ghanbari and Andrian Marcus. 2022. Patch Correctness Assessment in Automated Program Repair Based on the Impact of Patches on Production and Test Code. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA'22)* (Online). https://doi.org/10.1145/3533767.3534368

[16] Benny Godlin and Ofer Strichman. 2009. Regression Verification. In *Proc. of the 46th Design Automation Conference (DAC'09)* (San Francisco, CA, USA). https://doi.org/10.1145/1629911.1630034

[17] Tomasz Kuchta, Hristina Palikareva, and Cristian Cadar. 2018. Shadow Symbolic Execution for Testing Software Patches. *ACM Transactions on Software Engineering Methodology (TOSEM)* 27, 3 (2018), 10:1–10:32. https://doi.org/10.1145/3208952

[18] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Proc. of the 24th International Conference on Computer-Aided Verification (CAV'12)* (Berkeley, CA, USA). https://doi.org/10.1007/978-3-642-31424-7_54

[19] Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)* (Saint Petersburg, Russia). https://doi.org/10.1145/2491411.2491452

[20] Wei Le and Shannon D. Pattison. 2014. Patch Verification via Multiversion Interprocedural Control Flow Graphs. In *Proc. of the 36th International Conference on Software Engineering (ICSE'14)* (Hyderabad, India). https://doi.org/10.1145/2568225.2568304

[21] LLVM. 2024. clang-diff: A Tool for Comparing Abstract Syntax Trees. https://clang.llvm.org/. Accessed: 2024-09-08.

[22] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proc. of the 38th International Conference on Software Engineering (ICSE'16)* (Austin, TX, USA). https://doi.org/10.1145/2884781.2884872

[23] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: High-Coverage Testing of Software Patches. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'13)* (Saint Petersburg, Russia). https://doi.org/10.1145/2491411.2491438

[24] Phil McMinn, David Binkley, and Mark Harman. 2009. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Trans. Softw. Eng. Methodol.* 18, 3, Article 11 (June 2009), 27 pages. https://doi.org/10.1145/1525880.1525884

[25] Facundo Molina, Juan Manuel Copia, and Alessandra Gorla. 2024. Improving Patch Correctness Analysis via Random Testing and Large Language Models. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 317–328. https://doi.org/10.1109/ICST60714.2024.00036

[26] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. 2002. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *Proc. of the 11th International Conference on Compiler Construction (CC'02)* (Grenoble, France).

[27] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proc. of the Conference on Programing Language Design and Implementation (PLDI'07)* (San Diego, CA, USA). https://doi.org/10.1145/1250734.1250746

[28] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a Doubt: Testing for Divergences Between Software Versions. In *Proc. of the 38th International Conference on Software Engineering (ICSE'16)* (Austin, TX, USA). https://doi.org/10.1145/2884781.2884845

[29] Arindam Sharma, Daniel Schemmel, and Cristian Cadar. 2025. Reproduction Package for Article 'P³: Reasoning about Patches via Product Programs'. Zenodo. https://doi.org/10.5281/zenodo.16891174

[30] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering (ESEC/FSE'15)* (Bergamo, Italy). https://doi.org/10.1145/2786805.2786825

[31] Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified three-way program merge. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 165 (oct 2018), 29 pages. https://doi.org/10.1145/3276535

[32] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in Search-Based Program Repair. In *Proc. of the ACM Symposium on the Foundations of Software Engineering (FSE'16)* (Seattle, WA, USA). https://doi.org/10.1145/2950290.2950295

[33] Haoye Tian, Yinghua Li, Weiguo Pian, Abdoul Kader Kabore, Kui Liu, Andrew Habib, Jacques Klein, and Tegawendé F Bissyandé. 2022. Predicting patch correctness based on the similarity of failing test cases. *ACM Transactions on Software Engineering Methodology (TOSEM)* 31, 4 (2022), 1–30. https://doi.org/10.1145/3511096

[34] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *Proc. of the 40th International Conference on Software Engineering (ICSE'18)* (Gothenburg, Sweden). https://doi.org/10.1145/3180155.3180182

[35] Guowei Yang, Matthew B. Dwyer, and Gregg Rothermel. 2009. Regression model checking. In *Proc. of the IEEE International Conference on Software Maintenance (ICSM'09)* (Edmonton, Canada). https://doi.org/10.1109/ICSM.2009.5306334

[36] Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury. 2013. Expressing and checking intended changes via software change contracts. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (Lugano, Switzerland) *(ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/2483760.2483772

Received 2025-03-25; accepted 2025-08-12