Towards Automatic Patch Based Exploit Generation for Android C Libraries

Bhargava Shastry, Lee SeoungKyou

Computer Science Department
Rice University
Houston, USA

ABSTRACT

Managing the security of complex systems poses a great challenge. Incremental software patches are by far the most common means of plugging security holes: be it at the level of the operating system kernel, the middleware or the applications. However, the practice of system hardening via patch deployment assumes that the concerned patches are deployed uniformly across systems running a piece of software. This is far from reality: software vendors often stagger the deployment of patches for various reasons. This project is a step in the direction of building a static analysis framework that is able to automatically infer unsafe inputs from a patched program which when applied to an unpatched version of the same program cause the latter to misbehave in some way. We assume that there is an exploit directly related to unsafe inputs that are generated. We choose the Android mobile platform as a case study where our framework could be put to test.

1. INTRODUCTION

Contrary to the view that security patches enhance the security of a software system, recent work [5]—henceforth refered to as APEG —shows how the former can actually aid attackers in exploiting the very same vulnerabilities that were patched by exploiting the vulnerability on the unpatched version of the program. Note that this is nontrivial: a security patch need not reveal the vulnerability in a straightforward way; on average, it would take a seasoned hacker a few days of work at the very least to write an exploit to undermine an unpatched program. One would need to analyse the target program to list program variables that are effected by the patch and where they are used and ultimately craft an input that would attack the unpatched program.

A good example of staggered patch deployment is the Android mobile platform, where, due to fragmented variants of the platform (hardware and software), different phones seem to be running different versions of the Android operating system. The extent of fragmentation has been reported by OpenSignal [12]. It has to be noted that not only is the device (hardware) fragmented, but so is the entire software stack being used by a given device; the API-level (Android version) and the underlying Linux kernel version. Android's open nature has led to a proliferation of devices, each with subtly different hardware, running different variants of the software stack, not to talk about custom ROMs that are also available.

How is automatic patch-based exploit generation relavant to Android? The staggered nature of patch deployment is more of a problem for an open platform like Android than other closed mobile platforms like Apple's iOS. While APEG had to reverse engineer binaries to infer what were the changes effected by a patch, this step is completely unnecessary on Android whose source code is in the public domain.

Android's customisability is a double-edged sword—while users get to choose from multiple varied devices, app developers need to cater their offerings to each of these devices. More importantly, from a security point-of-view, vendors need to ensure that relevant security patches are deployed uniformly across all devices from time to time. This is far from what happens in practice.

Paper Overview.

This paper is organised as follows: Section 2 presents the problem statement, Section 3 discusses related literature. Section 4 introduces readers less familiar with program verification and its terminology to some fundamental terms and concepts in the area. Section 5 documents the implementation efforts during the course of this project; it mentions tools that we used and how they fit together by means of a suitable example. Subsequently, we present a simplistic program analysis framework that we wrote in the OCaml language, and our proposed design for a framework that is reproducing the work done in [5]. Section 6 highlights the deliverables of our course project and we conclude in Section 7. For the interested reader, Appendix A briefly talks about the internal workings of CIL (C Intermediate Language) and how it could be used for program analysis.

2. PROBLEM STATEMENT

Formally, APEG proposes techniques for the automatic patch-based exploit generation problem: given a program P, and a patched version of the same program, P', automatically generate an exploit for the potentially unknown vulnerability in P but patched in P'. Furthermore, the work targets input validation vulnerabilities only: vulnerabilities arising out of invalid inputs to the program. Hence, the attack can be modelled as a constraint satisfaction problem, with the constraint that an input fail the input sanitisation checks added in P' with an assumption that there is an exploit corresponding to a given input validation vulnerability. APEG argues that given that the delivery of patches is staggered over long time periods, there is a realistic time window in which unpatched systems are vulnerable to attack. Furthermore, if exploits

could be patch-driven in an automatic fashion, a hypothetical attacker would simply wait for patches to be available and subsequently exploit systems that have not been patched.

In this project, we seek to emulate the work done in [5] in a different setting. More precisely, our setting is different from theirs in the following ways:

1. APEG works on closed-source binaries (executables), while we only consider open-source software. This lets us focus on the problem of automatic exploit generation and not worry about reverse engineering a patch. Specifically, our target of attack would be userspace libraries (written in C language) in the Android software stack e.g., *bionic* libc, dalvik runtime libraries etc.

2. APEG proposes techniques based on dynamic program analysis only, static program analysis only and a mix of both, while we focus on static analysis only.

3. RELATED WORK

While our work itself is based on the problem that the APEG defines, there is a broad literature of prior work in this direction. In the following sections, we highlight past work in particular sub-areas.

3.1 Algorithmic trade-offs

The algorithms used in [5] are derived from previous works [4, 6]. Specifically, [6] notes that constriant formulae derived using the weakest pre-condition for a given vulnerability tend to be compact and faster to solve; this benefit positively influences (reduces) the number of control-flow paths that could be explored in turn resulting in fewer false negatives compared with other approaches to constraint formula generation.

A key concern in related literature is the tension between soundness (in a logical sense) of the proposed mechnisms and the efficiency with which they operate i.e., time to uncover as many control-flow paths as possible. Logical soundness means that there could be false positives but no false negatives whatsoever. For a static analysis approach to be sound, it needs to enumerate possible control-flow paths exhaustively, thereby leading to a a complexity that is exponential in the number of branches in a given program.

One can compromise on soundness by relaxing the requirement (for soundness) that all paths be explored; by exploring a subset of paths (possibly guided by complementary dynamic analysis), one could bring down the complexity of computing satisfiable constraint formulae from exponential to polynomial.

3.2 Automatic test case generation

Korat [3] is an automatic test case generation tool for Java programs. It relies on an annotated version of a Java method and generates test inputs to the method that satisfy the method's pre-conditions (constraints on input space) and uses the method's post-condition (specification on the method's output) as a test oracle to evaluate the correctness of the method's output for the automatically generated inputs.

A more targeted approach to test case generation is taken by the authors of [9]. In this paper, the problem is not to simply generate test cases for a given (C) program, but to actually generate test cases for those runs of the program in which a selected point in a given procedure is executed; this approach is termed goal-based test generation, the goal generation of inputs that implies execution of a certain control-flow path. The idea is closest to that in [5], where, inputs that fail a known check in the patched program are of interest.

4. BACKGROUND

We will use the example of a program (Algorithm 1) that computes the n^{th} power of a given integer to introduce the reader to pre and post conditions of a given program.

Algorithm 1 Calculate $y = x^n$

Require: $n \geq 0 \vee x \neq 0$
Ensure: $y = x^n$
1: $y \Leftarrow 1$
2: **if** $n < 0$ **then**
3: $X \Leftarrow 1/x$
4: $N \Leftarrow -n$
5: **else**
6: $X \Leftarrow x$
7: $N \Leftarrow n$
8: **end if**
9: **while** $N \neq 0$ **do**
10: **if** N is even **then**
11: $X \Leftarrow X \times X$
12: $N \Leftarrow N/2$
13: **else** $\{N$ is odd$\}$
14: $y \Leftarrow y \times X$
15: $N \Leftarrow N - 1$
16: **end if**
17: **end while**

Let Algorithm 1 refer to a program P, consisting of statements $S_1, S2, ..., S17$. The pre-condition, *pre* (prefixed by the keyword **Require** in Algorithm 1), of program P formalises conditions that are required to be satisfied by the program prior to its execution; in the present example, it is a first order logic formula in terms of program variables x and n. The post-condition of the program, *post* (prefixed by the keyword **Ensure**), formalises conditions that are required to be satisfied by the program after its execution. Note that *pre* and *post* are both expressions in first order logic i.e., a boolean predicate.

Program statements $S_1, ..., S_{17}$ may contain variables ($x, y \in V$), arithmetic and boolean expressions (E), and be composed of assignment statements ($V := E$) , if-then-else statements, and while loops.

The triple *pre* $\{S\}$ *post* is called a Hoare triple named after C.A.R. Hoare, a pioneer in the field of program verification. The Hoare triple is to be read as follows: If the pre-condition is satisfied by a program P consisting of statements $S_i \in S, i = 1, ..., n$, then, on execution of the statements in S, the post-condition will necessarily be satisfied should the program be correct and it terminates.

4.1 Weakest preconditions and strongest post conditions

We borrow introductory material and definitions of the weakest pre-condition and strongest post-condition in this section from [8]—a good survey of verification methods used today.

Consider two boolean predicates A and B. If the relation $A \implies B$ is true, then A is said to be stronger than B (since

A implies B); conversely B is said to be weaker than A.

The strongest post-condition of a program P with a pre-condition pre is denoted by $Sp\ \{P\}\ pre$. The strongest post-condition is the predicate obtained by transforming the pre-condition according to the interpretation of program statements in P. The strongest post-condition is termed so because, given a predicate $post$ that is true of any state obtained by interpreting the program P from an initial state where pre holds, $Sp\ \{P\}\ pre \implies post$.

The weakest pre-condition of a program with a post-condition $post$ is denoted by $Wp\ \{P\}\ post$. The weakest pre-condition is the predicate obtained by transforming the post-condition according to the interpretation of program statements in P in reverse order. The weakest pre-condition is weaker than the pre in that $pre \implies Wp\ \{P\}\ post$.

A relation between the Hoare triple $pre\ \{S\}\ post$ for a given program P and the weakest pre-condition Wp and the strongest post-condition Sp is that the Hoare triple holds if and only if $(Sp\ \{S\}\ pre) \implies post$ or if and only if $pre \implies (Wp\ \{S\}\ post)$.

5. IMPLEMENTATION

Our framework for static analysis is modeled for a subset of C language: it supports simple linear arithmetic—the use of + and × operator on program variables and constants.

5.1 Toolset

We use CIL [11] (C Intermediate Language), a program analysis and transformation tool for the C language. It is intended to be used for source to source transformations of "ugly" C code[1]. A key feature of CIL is that it is written entirely in OCaml (Objective Caml), a type-safe programming language that is primarily functional in nature, yet supports imperative and object-oriented programming styles.

Because CIL is written in OCaml and available as an add-on module for programs written in OCaml , it was natural for us to write our program analysis framework in OCaml . Furthermore, OCaml has been the language of choice in the program verification community [7, 10].

We quite heavily relied on CIL-template [1], a package containing sample programs written using CIL and OCaml, in the process of writing our framework. CIL-template not only provides one sample programs to get a feel of CIL but also build scripts that could be used to bootstrap the build process of our program analysis framework. This has been specially valuable for us during the course of this project.

We use Z3 [7], an SMT solver with a web interface[2], to solve constraints generated by our program analysis framework. We also employ Z3's advanced features like quantifier elimination to simplify the constraints that our analysis framework generates. It is important to note that Z3 is *not* a silver bullet oracle that can solve any constraint and return satisfiable values; it supports a limited subset of decision procedures, some of which cannot in general be decidable. [2] looks to extend the theories supported by Z3.

5.1.1 Program Analysis Example

CIL takes C-code as input, and serves a simple transformation from the code to C abstract Syntax Tree (AST). With this transformation, it traverses along the tree and executes various kinds of tasks defined in the user-written analysis program. For instance, CIL serves many use-cases, namely, data flow analysis, liveness analysis, ensuring program safety etc. Figure 5.2.1 describes the result of running a sample program (shown in Algorithm 1) to check if data values stored in program variables are even or odd. Uninitialized program variables are said to equal to "Bottom", in lay terms, *don't know*. Once program variables are initialized, the analysis program begins to pass judgements on the state of these variables. Note that, once instructions on line 5 are executed, the state of program variables becomes deterministic.

[1]The GCC C language testsuites presents a daunting challenge for compilers and parsers alike, see: `http://gcc.gnu.org/onlinedocs/gccint/C-Tests.html`

[2]Z3: `http://rise4fun.com/Z3`

```
bhargava@hp:~/software/cil-template$ ciltutcc --enable-tut3 -o tut3 test/tut3_m.c
test/tut3_m.c:5: (a, Bottom)
test/tut3_m.c:5: (b, Bottom)
test/tut3_m.c:5: (c, Bottom)
test/tut3_m.c:5: (d, Bottom)
test/tut3_m.c:6: (a, Odd)
test/tut3_m.c:6: (a, Odd)
test/tut3_m.c:6: (b, Even)
test/tut3_m.c:6: (c, Odd)
test/tut3_m.c:7: (c, Odd)
test/tut3_m.c:7: (c, Odd)
test/tut3_m.c:7: (d, Even)
test/tut3_m.c:7: (b, Even)
test/tut3_m.c:8: (b, Even)
test/tut3_m.c:8: (b, Even)
test/tut3_m.c:8: (d, Even)
test/tut3_m.c:8: (a, Even)
test/tut3_m.c:9: (a, Even)
test/tut3_m.c:9: (a, Even)
test/tut3_m.c:9: (a, Even)
bhargava@hp:~/software/cil-template$
```

Figure 1: Screenshot of program performing even-odd analysis of the program in Listing 1. The program has been borrowed from CIL-template. The analyzer here is inferring if the value stored in a program variable is even or odd at each step of the program. The value "Bottom" indicates that inference cannot be done.

Listing 1: C code: Input to even-odd analyser

```
1  #include <stdio.h>
2  int main()
3  {
4    int a,b,c,d;
5    a = 1; b = 2; c = 3; d = 4;
6    a += b + c;
7    c *= d - b;
8    b -= d + a;
9    if (a % 2) a++;
10   return 0;
11 }
```

5.2 Framework

While we have conceptualized the design of our framework for automatic exploit generation, it has not been fully implemented yet; what we have is a program analysis framework that is able to infer linear relations between program variables. This linear relations inference is based on computing the strongest post-condition (See Section 4) at each step in the program and solving for boolean constriants formulated solely on the less than equal to ($\leq$) arithmetic operator. We

intend to extend the present framework to be able to compute the weakest pre-condition as well. The latter will truly enable us to solve for inputs from a patched segment in the middle of a program.

Having said that, there are several other things to be taken care of before we can fully replicate the results in [5]: (1) interpret the effect of more complex instructions; at the moment we deal with simple arithmetic instructions involving the + and × operators, prominently pointers and arrays are not handled; (2) optimize the generated constraint formula by employing techniques such as program slicing. Furthermore, to be able to generate generic inputs to an unpatched program, one would need to employ an SMT (Satisfiability Modulo Theories) solver that supports the theory underlying the input;for instance if the input is an array, the chosen SMT needs to have decision procedures for the mathematical theory of arrays T_A, if the input is an integer, the solver would need to have a decision procedure for the theory of integers T_I and so on.

5.2.1 *Linear relations analysis*

The linear relations analysis framework is capable of inferring simple relationships between program variables. The framework has several limitations, namely, (1) It works for a limited subset of C language instructions and statements; prominently, pointers and arrays are not handled; (2) It does not work well for loops in general; we intend to make static analysis in the presence of loops more precise; (3) Finally, our framework is presently modeled on a simplistic constraint language consisting of boolean operators and the less than or equal to operator, $\leq$; for instance, at each point in a C program, our framework keeps a track of constraints in a first-order logical formula consisting of logical operators $\neg$, $\wedge$, $\vee$, and the $\leq$ arithmetic operator. Furthermore, we confine our analyis to linear arithmetic expressions (combination of + and × operators on program variables and constants). Notwithstanding the mentioned limitations, we believe linear relations analysis has given us experience in design of program analysis frameworks in general; also, the learning curve during the course of this project has been steep. We believe that the framework in its present form is a stepping stone towards a framework for the problem we set out to solve: automatic exploit generation.

The linear relations analysis framework works as follows: (1) It is written as a program in OCaml that imports CIL as a module; (2) It calls on CIL API's to obtain a control flow graph of an input C program and perform linear relations analysis on the variables in the program; (3) Specifically, at each program point, the program infers the linear relations that must hold, these are formulated as constraints involving the program variables and the $\leq$ operator; (4) Once the analysis program sees a check-point (labeled CHECK), the final constriant is spit out to the terminal; (5) One can then copy-paste the constriant in (4) into an SMT solver like Z3 that has a web interface and obtain the final output i.e., the linear relations between the variables.

Listing 2 shows a sample input for our linear relations analyzer and Listing 3 shows its output in the Z3 understood constraint language. Finally, we query the web-based Z3 SMT solver to obtain a solution to the boolean constraint. Figure 5.2.1 shows a screenshot of Z3's web-based interface solving the constriants generated by our program and printing out the end result at the bottom.

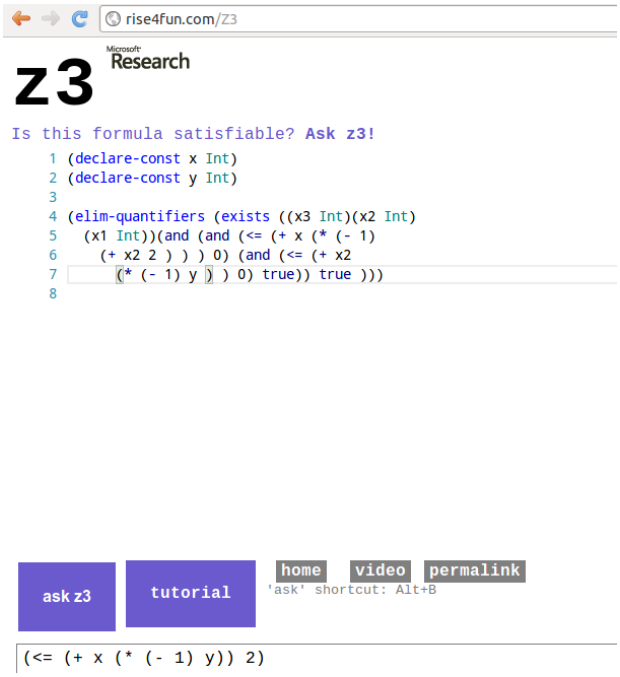

Figure 2: Screenshot of the web-based Z3 SMT solver printing out the resulting linear relation among program variables.

Listing 2: C code: Input to linear relations analyser

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main()
4  {
5  int x,y;
6  x = y;
7  x = x + 2;
8  goto CHECK;
9  CHECK: x = x + 1;
10 return 0;
11 }
```

Listing 3: Z3 constraint: Output of linear analyser

```
1 (declare-const x Int)
2 (declare-const y Int)
3
4 (elim-quantifiers (exists ((x3 Int)(x2 Int)
5 (x1 Int))(and (and (<= (+ x (* (- 1)
6 (+ x2 2 ) ) ) 0) (and (<= (+ x2
7 (* (- 1) y ) ) 0) true)) true )))
```

5.3 Design: Exploit generation

Sections 5.1 and 5.2 lay the groundwork for our work on exploit generation. In this section, we would like to discuss how we plan to implement a program analysis framework that is able to generate unsafe inputs (exploits) for unpatched programs by inferring relationships between program variables in a patched version of the same program. Our exploit generation framework would have access to the following programs: the original program P, and its patched version,

P'. We would like to explain how this works by means of an example from Android's libc codebase.

5.3.1 Memccpy Bug

Listings 4 and 5 refer to, respectively, the unpatched and patched versions of libc's memccpy program. In brief, as the comments in Listing 4 suggest, memccpy program takes 4 inputs: a destination address **dst**, a source address **src** from where a copy will be initiated, a search character **c**, and the number of bytes to be copied **n**. The memccpy program is supposed to copy either until it finds the characted **c** in the source address space, or until it copies **n** byets, whichever is earlier. It is supposed to return a null pointer, if and only if, the search character **c** is absent in **src** during copy. Otherwise, it is supposed to return the location of the character next to the (copied) **c** in **dst**.

Listing 4: C code: Input to linear relations analyser

```
1  #include <stddef.h>
2  #include <string.h>
3  /* Input:
4      src (mem location) of copy data
5      dst (mem location) where data should be
6           copied
7      c    search character "c"
8      n    number of bytes to be copied
9
10     Output:
11     NULL pointer if "c" not found in src
12     Otherwise
13     location of character next to "c" in src
14
15     Function Description:
16     Copy data from src to dst either:
17     1: Until (including) char "c" is found
18        in src
19     2: Or until "n" bytes have been copied
20 */
21 void *memccpy(void *dst, const void *src,
22               int c, size_t n)
23 {
24     char*       q     = dst;
25     const char* p     = src;
26     const char* p_end = p + n;
27     char        ch    = ~(char)c;
28     for (;;) {
29         if (ch == c || p >= p_end) break;
30         *q++ = ch = *p++;
31     }
32     if (p >= p_end){
33         return NULL;
34     }
35     return q;
36 }
```

However, the bug in the unpatched program is that the check for termination (line 32) does not cover the border case where **c** happens to be exactly the $\mathbf{n}^{th}$ character in **src**, in which case it incorrectly returns a null pointer. This to us, represents a sample input validation vulnerability leading to a null pointer return which could be possibly exploited. We flag this as a security bug. Note that the patched program fixes the bug by adding the mising check (line 15 in Listing 5).

Listing 5: C code: Input to linear relations analyser

```
1  #include <stddef.h>
2  #include <string.h>
3  void *memccpy(void *dst, const void *src,
4                int c, size_t n)
5  {
6      char*       q     = dst;
7      const char* p     = src;
8      const char* p_end = p + n;
9      char        ch    = ~(char)c;
10     for (;;) {
11         if (ch == c || p >= p_end) break;
12         *q++ = ch = *p++;
13     }
14     /* Patched check is (ch != c) */
15     if (p >= p_end && ch != c)
16         return NULL;
17
18     return q;
19 }
```

5.3.2 Proposed Framework

The proposed program analyser for automatic exploit generation would work as follows: (1) It takes as input a boolean constraint such that the set of inputs satisfying this constraint fail the check in P', the patched program; (2) It propogates this constraint backwards by computing the weakest precondition (see Section 4) at each step in the program; (3) It reaches the first line of P' at which point it gives us the constraints in terms of input variables. Subsequently, we take this constraint and input to an SMT solver like Z3 and Z3 in turn provides us sample inputs that satisfy the constraint; these are unsafe inputs that fail the patched check. Finally, we verify if the inputs solved for by Z3 are correct by using them on the unpatched program, P; if the inputs are unsafe indeed, it should trigger the buggy execution of P. Although this particular example might seem trivial, given that we propose a generic program analysis framework, we should be able to find more involved exploits using the same idea.

At present, we employ manual code scavenging aided by useful commands[3] to find out where to look for attack targets for the automatic exploit generation framework. Android's open-source code makes our task simpler compared to the binary emulation required in [5].

We now propose the design of our framework. Figure 5.3.2 is a flow diagram depicting the framework. As mentioned before, we give the patched program P' as input to our analyis framework; P' includes the code in P plus the added check to fix the known bug. The C-code analyser is written in the OCaml language and imports the CIL module (also written in OCaml). The analyser works its way up from the patched check, propogating the patched check (formulated as a boolean constraint) backward. Finally, it outputs a constraint in terms of input variables which when passed to an SMT solver like Z3, would give us sample input arguments to the program P for which the buggy execution is triggered.

6. DELIVERABLES

[3]e.g., **git whatchanged** shows what changed across multiple versions of a repository

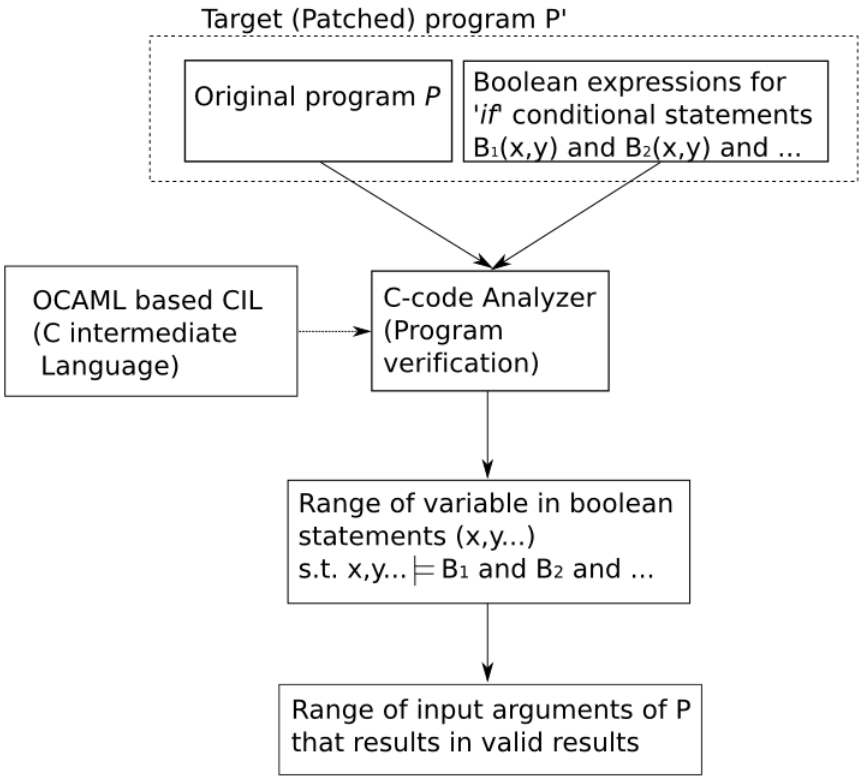

Figure 3: Proposed framework for automatic exploit generation

One of the main goals of this project was to develop a framework that is capable of performing static analysis on C code. We believe our project takes this goal forward. We have a proof of concept static analysis framework that is capable of inferring simple relationships between program variables. The present framework has several limitations, namely, (1) It works for a limited subset of C language instructions and statements; prominently, pointers are not handled; (2) It does not work well for loops in general; we intend to make static analysis in the presence of loops more precise; (3) Finally, our framework is presently modeled on a simplistic constraint language consisting of boolean operators and the less than or equal to operator, $\leq$; for instance, at each point in a C program, our framework keeps a track of constraints in a first-order logical formula consisting of logical operators $\neg$, $\wedge$, $\vee$, and the $\leq$ arithmetic operator. Furthermore, we confine our analyis to linear arithmetic expressions (combination of $+$ and $\times$ operators on program variables and constants).

Despite the limitations of our program analysis framework in its present form, we believe it is a good stepping stone towards building a static analysis framework that is able to reproduce the work in [5]. We have outlined the conceptual design of an automatic program generation framework in Section 5 and would like to idea forward and implement it.

7. CONCLUSION AND FUTURE WORK

We set out to build a program analysis framework that is able to automatically generate exploits as inputs to an unpatched program by analysis a patched version of the same program. At the moment, we have a working program analysis framework that is able to infer linear relations between program variables. Although this is far from what we set out to do, we believe that the experience we have gained building the linear relations framework will stand us in good stead and help us in taking the idea of automatic exploit generation forward.

During the course of this project, we have learnt to use tools like Z3, Dafny used in the program verification community today. We have also gained sufficient familiarity in writing programs in the OCaml language using CIL to aid us in program analysis.

We intend to extend the framework in its current form towards a framework that is able to solve the problem we set out to solve: generate unsafe inputs to unpatched versions of a program by analysing the patched version of the program. An interesting idea that could be explored in the future include cross-platform exploit generation: given a common code base for two different architectures, have the same tool generate (possibly) customised exploits for each of them.

8. REFERENCES

[1] Z. Anderson. cil-template. `https://bitbucket.org/zanderso/cil-template`.

[2] N. Bjørner. Engineering theories with z3. In *Proceedings of the First international conference on Certified Programs and Proofs*, CPP'11, pages 1–2, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international*

symposium on Software testing and analysis, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[4] D. Brumley, J. Newsome, D. Song, H. Wang, and S. Jha. Towards automatic generation of vulnerability-based signatures. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, SP '06, pages 2–16, Washington, DC, USA, 2006. IEEE Computer Society.

[5] D. Brumley, P. Poosankam, D. Song, and J. Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pages 143 –157, may 2008.

[6] D. Brumley, H. Wang, S. Jha, and D. Song. Creating vulnerability signatures using weakest preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium*, CSF '07, pages 311–325, Washington, DC, USA, 2007. IEEE Computer Society.

[7] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[8] M. Gordon and H. Collavizza. Forward with Hoare. In A. W. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, chapter 5, pages 101–121. Springer London, London, 2010.

[9] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, Mar. 1998.

[10] INRIA Saclay-Ile-de-France. Why3. `http://why3.lri.fr/`.

[11] G. Necula, S. McPeak, W. Weimer, B. Liblit, M. Harren, R. To, and A. Bhargava. CIL: Infrastructure for C Program Analysis and Transformation. `http://sourceforge.net/projects/cil/`.

[12] OpenSignal. Android fragmentation visualized. `http://opensignal.com/reports/fragmentation.php?`

APPENDIX

A. CIL: PROGRAM REPRESENTATION AND ANALYSIS

Parsing source code and constructing an Abstract Syntax Tree (AST) is one key feature of the CIL. CIL uses a Perl script to parse source code. An AST is a data structure that captures the semantics of a program by storing variables and language constructs like, *for*, *if*, or *while* systemically, as shown in Figure A.

AST is popular way to store source code systemically. By doing so, a probe (or a visitor in CIL) explores the source code tree structure algorithmically and predictably. Even more, users can customize traversal of the AST. Except some specific instances, most ASTs become a B-tree (or M-tree).

Once CIL parses C code and builds a tree, CIL's *visitor* is invoked to traverse the AST. One strong advantage of CIL is that one can customize the traversal of nodes in AST. CIL provides a basic visitor class with a default visitor method that does nothing; users can write customized visitor methods that inherit from this basic class. The tree visitor

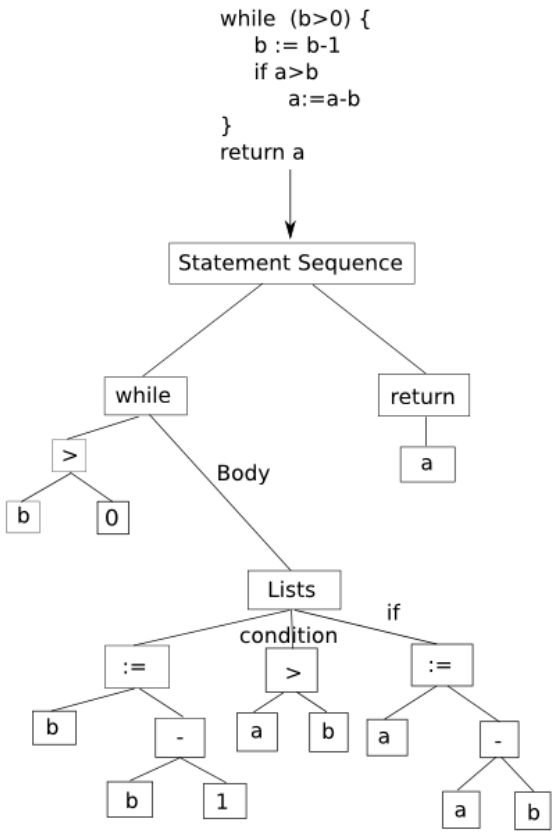

Figure 4: Example showing an abstract tree of a sample program. CIL's parsing method in *Cilly.pm* automatically parses the input C code into nodes in an AST comprising of variables (a or b), operators (:=, −, or >), and built-in functional statements (*while*, *return*, or conditional statement if).

class contains below four basic functions [11]:

1. SkipChildren: The visitor stops recursive visiting into child nodes in AST.
2. DoChildren: It guides the visitor to recursive traverse into child nodes.
3. ChangeTo x: It replace current node in AST with x.
4. ChangeDoChildrenPost(x, f): This function is similar to the "ChangeTo x" except that this is running the function "f" base on the result of the running on x's children.