Efficient Incremental Static Analysis Using Path Abstraction

Rashmi Mudduluru and Murali Krishna Ramanathan

Indian Institute of Science, Bangalore, India
{mudduluru.rashmi,muralikrishna}@csa.iisc.ernet.in

Abstract. *Incremental* static analysis involves analyzing changes to a version of a source code along with analyzing code regions that are semantically affected by the changes. Existing analysis tools that attempt to perform incremental analysis can perform redundant computations due to poor abstraction. In this paper, we design a novel and efficient incremental analysis algorithm for reducing the overall analysis time. We use a *path abstraction* that encodes different paths in the program as a set of constraints. The constraints encoded as boolean formulas are input to a `SAT` solver and the (un)satisfiability of the formulas drives the analysis further. While a majority of boolean formulas are similar across multiple versions, the problem of finding their equivalence is *graph isomorphism complete*. We address a relaxed version of the problem by designing efficient *memoization* techniques to identify equivalence of boolean formulas to improve the performance of the static analysis engine. Our experimental results on a number of large codebases (upto 87 KLoC) show a performance gain of upto 32% when incremental analysis is used. The overhead associated with identifying equivalence of boolean formulas is less (not more than 8.4%) than the overall reduction in analysis time.

1 Introduction

The adoption of static analysis tools for bug detection [4,20] in the software development cycle has increased significantly in the past decade [5]. These tools typically analyze the source code to find different kinds of bugs including null pointer dereferences [9,12], resource leaks [18], concurrency bugs [5] and security flaws [19]. Software testing can be ineffective because it requires significant manual effort to write good test cases to detect complex bugs. In contrast, static analysis tools [13,20] can detect deep interprocedural defects on even rare execution paths *automatically* resulting in their increased use in software organizations. However, the time taken for static analysis to analyze the entire codebase can be quite high. This has resulted in the design of a number of techniques to reduce the analysis time [13,2].

In many organizations, each build goes through an automated software testing process. If static analysis is integrated into the build system and the process is time consuming, it can result in the unavailability of fresh testable binaries. To address this, if the analysis is decoupled with the build process, the defects

S. Gnesi and A. Rensink (Eds.): FASE 2014, LNCS 8411, pp. 125–139, 2014.

eventually found by the static analysis will anyway *invalidate* the automated testing. This is because the defects detected by the analysis need to be fixed and automated testing needs to happen subsequently. Therefore, a practical mode of using static analysis is to integrate them into the nightly build [13]. As a result, the problem of improving the efficiency of static analysis has gained significance. Many approaches for improving the efficiency of static analyses [20,13,2] exploit the underlying parallelism in analyzing the code. For example, some functions can be analyzed concurrently to reduce the overall analysis time.

A state of the art static analysis tool that is based on the above paradigm is `SATURN` [20]. It is a path sensitive, interprocedural and a highly scalable static analysis tool that has its foundations on boolean satisfiability and models the programs being analyzed as a set of boolean constraints where the entire program behaviour including path information is encoded by the constraints. Each program point is represented as a boolean combination of these constraints. To check a property at a given program point, `SATURN` converts the boolean formula representing the program point into its conjunctive normal form and invokes a `SAT` solver on the formula. Thus, the problem of detecting bugs is reduced to that of checking the satisfiability of the boolean constraints at various program points. SATURN also has a parallel implementation of their analysis which distributes independent functions across multiple cores.

While parallelization of static analysis is a necessary first step in improving its scalability, there are other avenues for reducing the analysis time. An unexploited avenue of improvement is the reanalysis of unchanged source code. Programmers typically make only a few source code changes for every revision. *Incremental analysis* involves analyzing the modified parts of the source code and its dependencies. In practice, McPeak *et al* [13] show that incremental analysis is very effective in reducing the overall analysis time on revisions of the codebase.

We observe that the constraints, generated by SATURN, across successive versions are mostly the same because a majority of the codebase is unchanged. A minor change in a function only results in modification to a subset of these constraints. Because these constraints are not necessarily coupled to a function, many boolean constraints solved by the `SAT` solver are common across multiple functions within the same version of the codebase.

Recently, incremental static analysis techniques have been proposed for different applications [22,17,11,21,13]. For example, McPeak *et al* [13] propose an incremental static analysis tool for bug detection that abstracts a function in terms of a work unit. If the work unit is unchanged, then the result of analyzing the work unit from the previous run of the static analysis is used. We believe that this abstraction is coarse-grained and has many limitations. Firstly, a minor change in the work unit which does not necessarily affect the static analysis results can indeed cause a reanalysis of the work unit. Secondly, the work unit abstraction results in all paths within a function (including semantically independent paths) being coupled together and therefore prevents reusability of analysis results on unaffected paths. Finally, this abstraction does not leverage

the potential reusability of results within the same version of the codebase. In this paper, we explore the possibility of using a *path abstraction* for designing incremental analysis which is fine grained and does not suffer from the above limitations.

We leverage the path abstraction properties of `SATURN` to build an incremental static analysis tool, named `iSATURN`, which addresses the limitations of work unit abstraction for performing incremental analysis. When `iSATURN` analyzes the source code initially, it *memoizes* the boolean constraints that are input to the `SAT` solver and the associated results from the solver. The number of symbols in a boolean formula can be significantly high. To reduce the overhead with hashing the entire boolean formula, we pick a subset of the symbols *deterministically*, hash it and use it as a key to identify the formula and its results. We store the memoized results in a highly efficient key-value database. In any subsequent analysis, the solver is not invoked if the database lookup for the boolean formula succeeds. The reduction in the number of invocations to the solver reduces the overall analysis time. Moreover, as these formulas are function agnostic, memoized results of boolean formulas solved earlier in the run can be reused appropriately.

We have implemented our approach on top of the `SATURN` analysis framework. We use `BerkeleyDB` [16] for storing the memoized results. Our experiments on multiple versions of large codebases show that the path based abstraction for performing incremental analysis reduces the overall analysis time upto 32%. More importantly, the results observed with the incremental static analysis is exactly the same as the results obtained using a full analysis.

We make the following technical contributions in this paper:

1. We leverage the path based abstraction of static analysis tools to design a novel and scalable incremental static analysis.
2. We implement our approach on top of the `SATURN` analysis framework to analyze `C` programs.
3. We analyze multiple versions of large (upto 87 KLoC) open source codebases showing upto 32% reduction in analysis time when compared to the overall analysis time of `SATURN`.

2 Motivation

In this section, we motivate our problem by using examples to illustrate the inefficiencies with existing static analysis engines. Figure 1 shows the implementation of function `gunzip_file` in two successive versions of `busybox`. Assume the application of a static analysis tool (e.g., `SATURN`) to identify null pointer dereferences [1]. Consider the program point corresponding to line number `135` in `busybox v0.60.4`. At this point, `SATURN` generates boolean constraints to check the satisfiability of conditions, `unzip(in_file, out_file) == 0` and `path == NULL`. The first condition is a conditional that is present in the code and the second condition corresponds to the static analysis check to identify the feasibility of a null pointer at that point. Now, consider the program point corresponding to

busybox v0.60.4

```
74 static int gunzip_file (const char *path, int flags)
75 {
    ....
133 if (unzip(in_file, out_file) == 0) {
134 /* Success, remove .gz file */
135   delete_path = path;
136   if (flags & gunzip_verbose) {
137       fprintf(stderr, "OK\n");
138       }
    ....
  }
```

busybox v0.60.5

```
74 static int gunzip_file (const char *path, int flags)
75 {
    ....
133 if (unzip(in_file, out_file) == 0) {
134  /* Success, remove .gz file */
135  if ( !(flags & gunzip_to_stdout ))
136    delete_path = path;
137    if (flags & gunzip_verbose) {
138        fprintf(stderr, "OK\n");
139        }
    ....
  }
```

Fig. 1. Illustrative Example: Function `gunzip_file` in two versions of `busybox`

line number `136` in `busybox v0.60.5` (right side of Figure 1). For this version, the boolean constraints generated to check the satisfiability of the conditions include `unzip(in_file, out_file) == 0`, `!(flags & gunzip_to_stdout)` and `path == NULL`. Obviously, the boolean constraints at the program point under consideration will *differ* for these two versions. Interestingly, the boolean constraints will *only* differ on paths that have either changed syntactically or are semantically dependent on the changed syntactic paths. In the above example, out of the 94 boolean formulas in the function that are checked for satisfiability, only 39 (40%) boolean formulas in the newer version differ from the original set. We observe that eliminating the redundant computation of checking boolean satisfiability can improve the performance of static analysis. Existing static analysis tools [13,20] will reanalyze the entire function. Our design of a more fine grained incremental analysis technique enables our tool, `iSATURN`, to effectively eliminate redundant computations.

Not surprisingly, our mechanism for improving the performance of static analysis also benefits analyzing the same version. Consider the code fragments for the functions `md5_get_result` and `md5_init` from `bftpd v3.1` shown in Figure 2. In line `244` of `md5_get_result`, the first parameter, `md5_p` to the function is dereferenced and there is a corresponding boolean formula associated with this

```
239 static void md5_get_result(const md5_t *md5_p, void *result)
240 {
241   md5_uint32    hold;
242   void *res_p = result;
243
244   hold = SWAP(md5_p->md_A);
245   memcpy(res_p, &hold, sizeof(md5_uint32));
246   res_p = (char *)res_p + sizeof(md5_uint32);
    ....
  }
```

```
282 void md5_init(md5_t *md5_p)
283 {
284   md5_p->md_A = 0x67452301;
285   md5_p->md_B = 0xefcdab89;
    ....
  }
```

Fig. 2. Illustrative Example: Commonality across two functions in the same version (`v3.1`) of `bftpd`

program point. Similarly, there is a boolean formula associated with dereferencing the first parameter `md5_p` of function `md5_init` at line `284`. In our analysis, we observe that these two boolean formulas are exactly the same as these dereferences happen unconditionally and therefore the `SAT` solver need not be invoked twice unnecessarily. We observe similar *c*ollisions of boolean formulas within the same version of the program across multiple benchmarks.

3 Background

For completeness, we provide a brief overview of `SATURN` [20]. The framework provided by `SATURN` translates a C program into a set of boolean constraints. It has rules for modelling common C constructs like integers, pointers, conditionals, etc. For example, integers are encoded as their binary representations. Consider the statement: (c == 10)? s1 : s2; Here, statement s1 is executed if c has the value 10. So the boolean constraint that encodes the condition under which s1 is executed is (B(c) == 1010), where B(c) is the binary representation of the integer c. For s2, the constraint will be $\neg(B(c) == 1010)$.

Any property checker (e.g., null pointer analyzer, leak detector, etc.) is encoded as a set of finite state properties and plugged into this framework. This is accomplished by expressing the property checkers as a set of inference rules. For example, one such inference rule could state that given the premises (i) a pointer variable X points to another variable Y and (ii) at program point P, expression E evaluates to X hold, the conclusion "E evaluates to Y" holds [1]. If the property being checked is aliasing, then the function summary is inferred by checking the satisfiability of constraints obtained in conjunction with these properties and the primitive constraints generated by `SATURN`. Consequently, bug detection is reduced to checking if the error state is reachable. This checking is done with the help of SAT queries [1].

Interprocedural analysis is achieved by using function summaries. Function summaries represent the state of the function at its exit with respect to the property being checked. For example, if an argument is dereferenced within a function, and the property that is under consideration is null pointer dereference, then the function summary corresponding to the function will carry the fact that the argument to the function is dereferenced. For a more detailed exposition of `SATURN`, we refer the reader to [20].

To illustrate the working of how the null analysis in `SATURN` detects bugs, consider the example shown in Figure 3. `SATURN` performs bottom up analysis [13] by default. In this example, the function `bar` is analyzed before `main`. The boolean formulas generated by the null analysis for the function `bar` encode the constraints shown in Figure 4. The constraints on the left correspond to the true branch of the `if` condition and those on the right correspond to the `else` branch. When the function `bar` is analyzed, all these boolean formulas are satisfiable. Eventually, when `main` is analyzed, the function summary for `bar` is available. Among other facts, this summary contains the fact that if `flag` is not equal to `0`, then the variable `var` is certainly dereferenced. This information

```
char *p1, *p2;
void bar(int flag)
{
  char var;
  var = (!flag? *p1 : *p2);
}
```

```
void main(int args)
{
  p2 = 0;
  bar(1);
}
```

Fig. 3. Illustrative example to describe bug detection in `SATURN`

1. arg(0) == 0 & p1 == 0
2. arg(0) == 0 & p1 != 0
3. arg(0) == 0 & p2 != 0

1. arg(0) != 0 & p1 != 0
2. arg(0) != 0 & p2 == 0
3. arg(0) != 0 & p2 != 0

Fig. 4. Boolean constraints for function bar() from Figure 3

along with the currently known facts, `p2 = 0` and `flag = 1` results in the null dereference bug being detected.

4 Design

In this section, we describe the overall architecture of our approach and subsequently describe the incremental analysis in detail.

4.1 Architecture

Figure 5 shows the overall architecture of `iSATURN`. `CIL` [15], the frontend `C` parser, parses the program and generates abstract syntax trees (ASTs) [13]. These ASTs are encoded as a set of predicates representing program relations. A database is created for each function that stores these predicates as a key value pair. These databases are known as syntax databases [1].

`SATURN` [20] consults the syntax databases and generates boolean constraints depending on the property being checked. In doing so, the path information is also encoded in the boolean constraints. While generating constraints, `SATURN` uses the predicates in the syntax databases to infer conditions under which a

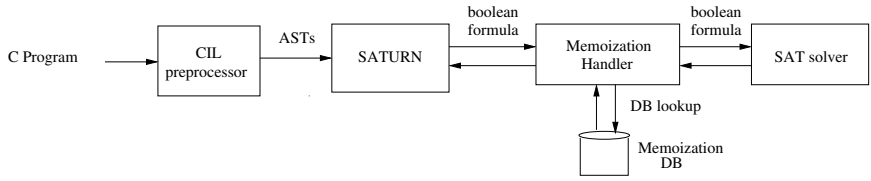

Fig. 5. Architecture of iSATURN

program point is reachable, thereby generating precise path information. SATURN thus performs an analysis of the memory model to generate path sensitive boolean formulas. These formulas are passed to a SAT solver [8] in their conjunctive normal forms. The results from the SAT solver help SATURN in generating the function summaries and error reports. The summary of a function represents all the states of the function at its exit with respect to the property being checked. Based on the results from the SAT solver, SATURN infers the predicates that are feasible for a function and generates function summaries [1].

In our tool, iSATURN, we introduce a memoization database before the invocation of the SAT solver. We choose BerkeleyDB [16], a high performance key value store for this purpose. The memoization database tracks the boolean formulas solved by the SAT solver and the corresponding results. Therefore, in iSATURN, we first check whether the results for a boolean formula are already present in the memoization database. We define the presence of the boolean formula as a *hit* and the absence as *miss*. In case of a hit, the SAT solver is not invoked and the results from the database are returned rightaway. Otherwise, the SAT solver is invoked as usual. We find that across successive versions, the number of calls to the SAT solver decreases significantly due to the repetitive nature of the boolean formulas being generated.

4.2 Equivalence of Boolean Formulas

We hypothesize that elimination of redundant invocations of the SAT solver to check the satisfiability of the boolean formulas can help improve the performance of the static analysis engine. By reusing the results of the solved boolean formulas, the analysis time can be decreased significantly. While stated simply, there are significant challenges to bringing this idea to fruition. To reuse the results, we need to identify the equivalence of boolean formulas.

Boolean Formula Structural Equivalence Problem: For two boolean formulas,

1. $\phi_1 = (a_{11} \vee a_{12} \vee ....a_{1i}) \wedge (a_{21} \vee a_{22} \vee ....a_{2j}) \wedge .... \wedge (a_{n1} \vee a_{n2} \vee ....a_{nk})$
2. $\phi_2 = (b_{11} \vee b_{12} \vee ....b_{1l}) \wedge (b_{21} \vee b_{22} \vee ....b_{2m}) \wedge .... \wedge (b_{n1} \vee b_{n2} \vee ....b_{np})$

ideally the two CNF formulas are equivalent, if there exists some permutation of the clauses and some permutation of the literals within these clauses resulting in a one to one mapping of variables in these formulas. More specifically, let the clauses of ϕ_1 be denoted as cl_1, cl_2, ..., cl_n. Now consider the following transformations on ϕ_1:

1. $\phi'_1 = cl'_1 \wedge cl'_2 \wedge ... \wedge cl'_n$, where $(cl'_1, cl'_2..., cl'_n)$ is a permutation of $(cl_1, cl_2, ..., cl_n)$
2. $\phi''_1 = (a''_{11} \vee a''_{12} \vee ....a''_{1l}) \wedge (a''_{21} \vee a''_{22} \vee ....a''_{2m}) \wedge ... \wedge (a''_{n1} \vee a''_{n2} \vee ....a''_{np})$, where $(a''_{i1}, a''_{i2}, ..., a''_{ik})$ is a permutation of literals in cl'_i

If the variables in ϕ''_1 can be renamed such that the resulting formula becomes equal to ϕ_2, then the two formulas ϕ_1 and ϕ_2 are structurally equivalent.

Any truth assignment to the variables of ϕ_1 can also be applied to the corresponding variables in ϕ_2. For example, the following two boolean CNF formulas are structurally equivalent:

1. $\psi_1 = (a \vee \neg b \vee c) \wedge (d \vee a \vee b)$
2. $\psi_2 = (z \vee w \vee x) \wedge (w \vee y \vee \neg x)$

Permuting the clauses of ψ_1, we get $\psi_1' = (d \vee a \vee b) \wedge (a \vee \neg b \vee c)$. Further, permuting the literals of the second clause in ψ_1' results in $\psi_1'' = (d \vee a \vee b) \wedge (a \vee c \vee \neg b)$. There is a one to one correspondence between the variables of ψ_1'' and ψ_2 (i.e) (a, b, c, d) corresponds to (w, x, y, z). Any satisfying truth assignment of (a, b, c, d) for ψ_1 can also be applied to (w, x, y, z) in ψ_2 respectively.

Theorem 1. *The boolean CNF formula structural equivalence problem is graph isomorphism complete [3].*

Proof. The problem of determining equivalence of CNF formulas is reducible to that of graph isomorphism in polynomial time [3].

If we make the approach of considering equivalence of formulas more conservative by foregoing the permutation of clauses and literals in defining the equivalence of CNF formulas, then the approach has a polynomial time complexity. More specifically, for two CNF formulas,

1. $\phi_3 = (a_{11} \vee a_{12} \vee ....a_{1i}) \wedge (a_{21} \vee a_{22} \vee ....a_{2j}) \wedge .... \wedge (a_{n1} \vee a_{n2} \vee ....a_{nk})$
2. $\phi_4 = (b_{11} \vee b_{12} \vee ....b_{1i}) \wedge (b_{21} \vee b_{22} \vee ....b_{2j}) \wedge .... \wedge (b_{n1} \vee b_{n1} \vee ....b_{nk})$

the formulas are said to be equivalent if there exists a one to one mapping of variables such that the literals in ϕ_3 can be renamed as those in ϕ_4 at their corresponding positions. This conservative technique gives fewer number of hits than the original boolean CNF formula equivalence problem. In other words, it may miss some formulas that are actually equivalent because of the permutations of the clauses and the literals within. As a result, ψ_1 and ψ_2 given above would not be considered equivalent in this approach. However, in this approach, the following two boolean formulas are considered equal:

1. $\psi_3 = (a \vee \neg b \vee c) \wedge (d \vee a \vee b)$
2. $\psi_4 = (w \vee \neg x \vee y) \wedge (z \vee w \vee x)$

ψ_3 and ψ_4 are considered equivalent in this approach because there is a one to one correspondence between their clauses, and within each clause, each variable of ψ_3 can be mapped to the variable at the corresponding position in ψ_4.

Nevertheless, even this conservative approach does not necessarily scale because it involves canonical renaming of variables and their comparison. In order to rename a variable, we need to maintain a mapping of the variables' names and their new names. For each variable, we check if it has already been renamed. If not, we assign it a new name in a canonical manner. This has to be done for every literal in every clause. This results in a quadratic complexity in the total number of literals in the formula.

Therefore, we consider the most practical approach to compare the formulas. Here, two formulas are considered equivalent only if they are exactly the same. This approach reduces the number of formulas that are syntactically equivalent and results in more invocations of the `SAT` solver than is necessary. Surprisingly, our experimental results shows that the number of hits even with this technique is significantly high and helps improve the performance of static analysis.

5 Implementation

Algorithm 1. Memoization Handler

Input: $\phi = (c_{11} \vee c_{12} \vee ....c_{1i}) \wedge (c_{21} \vee c_{22} \vee ....c_{2j}) \wedge .... \wedge (c_{n1} \vee c_{n2} \vee ....c_{nk})$
Output: SAT or UNSAT

1: $\text{Num}_c(\phi)$: number of clauses in ϕ
2: $\text{Num}_l(\psi)$: number of literals in ψ
3: $H(i_1, i_2, .., i_m)$: MD5 hash of concatenation of $i_1, i_2, ..., i_m$
4: **if** $\text{Num}_c(\phi) < K$ **then**
5: key ← H(...H(H($c_{11}, c_{12}, ..., c_{1i}$),($c_{21}, c_{22}, ..., c_{2j}$)),....,($c_{n1}, c_{n2}, ..., c_{nk}$))
6: **else**
7: key ← ""
8: index ← 1
9: **for** each clause ψ in ϕ **do**
10: pos ← index mod ($\text{Num}_l(\psi)$)
11: key ← H(key,H(literal at position **pos** in ψ))
12: index ← index + 1
13: **end for**
14: value ← memdb_lookup(key)
15: **if** value is found **then**
16: return the value
17: **else**
18: result ← SAT_solver(ϕ)
19: memdb_store(key,result)
20: return result
21: **end if**
22: **end if**

In our implementation, we use `BerkeleyDB` [16], a highly efficient key value store to memoize the formulas and their results. If the formulas include a lot of literals, memoizing the boolean formulas as strings is highly inefficient. In practice, because SATURN encodes programs as boolean constraints and maintains precise path information down to the level of bits, these boolean formulas tend to be very large. For example, in our experimental setup, for benchmarks of about 7 KLoC, we observe that the maximum number of clauses is 7000 and the maximum number of literals within each clause is 100 approximately.

We address the above problem by employing an efficient hashing technique to store the formulas. Given a boolean formula $(c_{11} \vee c_{12} \vee ....c_{1i}) \wedge (c_{21} \vee c_{22} \vee ....c_{2j}) \wedge ....\wedge (c_{n1} \vee c_{n2} \vee ....c_{nk})$, we store $H(..H(H(c_{11}, .., c_{1i}), (c_{21}, .., c_{2j})), .., (c_{n1}, .., c_{nk}))$

as the key, where $H(a, b)$ denotes the hash of concatenation of a and b. We employ the `MD5` algorithm [7] for hashing which ensures that every key is 128 bits long.

To optimize the implementation further and to leverage the large size of the boolean formulas, we also employ a *deterministic sampling* approach to generate the key for every boolean formula. Instead of using the complete formula to generate the hash, the hash is based on picking a predetermined literal from each clause. The hash function used is $H(...H(H(c_{1a}), c_{2b}), ..., c_{nk})$, where n is the number of clauses in the formula, c_{ij} denotes the j^{th} literal of i^{th} clause, where $j = i \mod f(i)$ and $f(i)$ is the number of literals in i^{th} clause. The complete algorithm is shown in Algorithm 1.

In Algorithm 1, for formulas with number of clauses less than K, lines 4 and 5 generate the key for database lookup by hashing the entire SAT formula. For the remaining formulas, lines 9 to 13 generate the key by hashing the sampled literals from each clause in the SAT formula. The rest of the algorithm describes how the generated key is used in database lookup. Upon a hit, the result stored in the DB is returned. Otherwise, the SAT solver is invoked and the result is returned after storing it against the generated key in the memoization DB.

Though the deterministic sampling may result in two unequal formulas being matched, the probability of such a scenario is very low. More specifically, the probability of two distinct formulas being considered equivalent due to sampling will happen if the sampled literal from each clause matches. The probability of two distinct formulas matching because of deterministic sampling is $\Pi_{i=1}^{n} 1/f(i)$. For sufficiently large n and $f(i)$, we deploy the sampling technique and for smaller values, we use the complete approach. In our experimental results, we also empirically verify across a number of benchmarks that deterministic sampling does not result in two distinct formulas being considered equivalent.

6 Experimental Results

We conduct our experiments on an Ubuntu 12.04 desktop equipped with 3.5 GHz Intel core i7 processor and 16 GB RAM. We use `BerkeleyDB` version 5.3.21 to store the SAT formulas and their results. We use the null pointer dereference analysis to show the scalability of our approach. We believe that the conclusions drawn from the experiments can be extended to other interprocedural analyses as well. We use five open source benchmarks for our experiments. `busybox` is a collection of utilities for embedded systems, `openssh` uses the SSH protocol to provide secure communication sessions over a network, `gzip` is used for compression and decompression of files and `bftpd` is a flexible FTP server. We use two different versions of `gzip` (`gzip(A)` and `gzip(B)`) to show that the performance of our approach is also dependent on the kinds of changes made across the versions.

Table 1 provides the list of benchmarks used and the statistics associated with the benchmarks including the number of lines of code and the number of formulas that need to be solved by the static analysis engine. The number of lines of code varies from `5` KLoC to `88` KLoC approximately. The number of formulas varies

Table 1. Benchmarks used in our experiments and associated statistics

Benchmark	Version 1	Version 2	KLoC		Number of formulas	
			version 1	version 2	version 1	version 2
busybox	0.60.4	0.60.5	75.5	75.8	48198	48388
openssh	20130823	20130830	87.7	87.9	106305	106305
bftpd	3.1	3.2	5.2	5.2	9638	9704
gzip(A)	1.2.4	1.2.4a	8.0	8.0	51363	51363
gzip(B)	1.3.2	1.3.3	9.2	9.2	56210	56210

Table 2. Comparison of performance between SATURN and iSATURN

Benchmark	version 1			version 2			Memory overhead percentage*
	T_S (secs)	T_I (secs)	Overhead percentage	T_S (secs)	T_I (secs)	Improvement percentage	
busybox	333	358	7.5	337	228	32	0.83
openssh	661	717	8.4	664	455	31.4	1.11
bftpd	65	66	1.5	65	58	10.7	0.22
gzip(A)	2811	2987	6.2	2891	2069	28.4	0.66
gzip(B)	7071	7309	3.3	7212	6734	6.6	0.7

T_s : Analysis time with SATURN.
T_I : Analysis time with iSATURN.
*Memory overhead of iSATURN over SATURN.

from `9638` for `bftpd` to `106305` for `openssh`. In scenarios where the number of formulas is the same across versions (Example: `openssh`, `gzip(A)`, `gzip(B)`), we will show later that they are not necessarily the same set of formulas.

Table 2 presents a comparison of `iSATURN` and `SATURN` in analyzing the five benchmarks. For each version of the benchmark, we analyze it using both the tools. The choice of SAT formulas that are sampled and not cached completely by `iSATURN` depends on the structure of the formula being considered. Specifically, while converting the SAT formulas into their CNF forms and simplifying them, SATURN has a way of putting the clauses into either one group or two groups. Typically, when the clauses are put into two groups, the number of clauses in the first group is very large and we sample the literals from the clauses of this group. We run the analysis to fixpoint or 20 iterations, whichever completes earlier. We also do not consider analyzing functions[1] that timeout due to implementation issues in `SATURN` itself. In the first version, there will be overhead associated with `iSATURN` because the results pertaining to the boolean formulas are being stored in the database. This overhead varies from 1.5 to 8.4%. Subsequently, when the second version of the benchmark is analyzed by both the tools, `iSATURN` shows a performance improvement of upto `32`% for `busybox`. This performance gain can be obtained across multiple runs of the program on the same version or any subsequent version. We also observe that in `bftpd`, the size of the SAT

[1] The number of formulas that timeout is less than 3.5%.

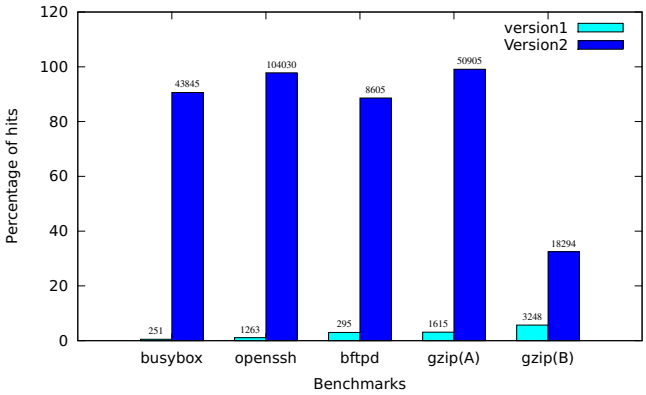

Fig. 6. Percentage of hits across versions. The number on top of the bars represents the actual number of hits.

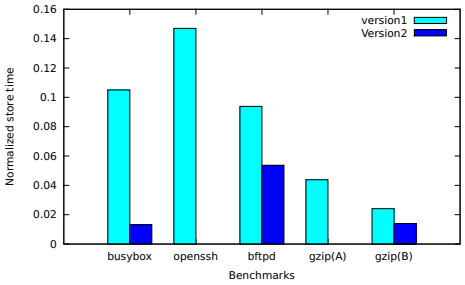

Fig. 7. Normalized store time

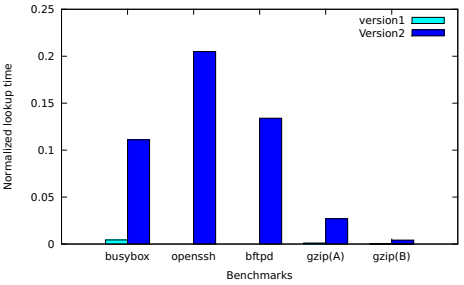

Fig. 8. Normalized lookup time

formulas does not exceed 2502 clauses and it has the least overhead during the analysis in the first version (1.5%). On the other hand, analysis of `openssh` generates formulas with the number of clauses as high as 22839 but the overhead is relatively small (8.4%). The last column in table 2 shows that the memory overhead incurred by `iSATURN` over `SATURN` is negligible.

Figure 6 shows the benchmarks on the X axis and number of hits on the Y axis. Recall that a hit happens whenever the formula that needs to be solved has its results stored in the database. While analyzing successive versions of the codebase, the number of hits increases significantly. With increase in the number of hits, the reduction in the overall analysis time is prominent. Furthermore, we also observe that the database lookup succeeds even when analyzing the same version. For example, while analyzing the first version of `gzip(B)`, we notice a non trivial (3248) number of hits.

Figure 7 compares the normalized time of storing the SAT formulas and their results across versions. The time to store the formulas and the results does not exceed `15`% of the overall analysis time and is negligible in many cases.

Figure 8 shows the normalized lookup time across versions. The lookup time also does not exceed 20% of the overall analysis time. Using a more efficient key value store can reduce these overheads even further.

The results show that the techniques used here can be employed to improve efficiency of applications where a number of SAT formulas need to be solved and many of them are equivalent as described here. However, in applications where the precise equivalence of SAT formulas is essential, the sampling technique should not be employed.

7 Related Work

McPeak *et al* [13] propose a solution for the problem of incremental static analysis and provide a solution. One of the fundamental limitations of their approach is that the incremental analysis is based on a coarse grained work unit. Therefore, even a minor modification to a function which does not necessarily change the outcome of static analysis can result in the function being re-analyzed. In other words, their approach does not fully exploit semantically equivalent paths. Furthermore, even while analyzing the same version, the avenues for reducing redundancy is not explored comprehensively. Our approach is complementary to their approach. For functions where the approach of [13] suggests a reanalysis, `iSATURN` can be used to reduce the redundancy.

Recently, memoization has been applied to improve efficiency of symbolic execution and model checking. Yang *et al* [22] propose memoized symbolic execution, where a trie data structure is used to represent symbolic paths generated during a symbolic run. Subsequent symbolic runs of the program use the information stored in the trie to avoid re-execution of unaffected paths. Person *et al* [17] describe techniques to direct the symbolic execution using changes across different versions of a program. Path conditions affected by code changes are computed and are used to explore execution paths in the affected parts of the program. Lauterburg *et al* [11], show how state space exploration can be made incremental by memoizing information about states that are unaffected by changes across program versions. Similarly, Yang *et al* [21] propose regression model checking where data from previous versions helps to avoid checking some state spaces that are safe in the current version. It uses test suites to identify paths in a CFG that are modified in the current version and thereby prunes states that are safe. Our approach differs from these techniques as it aims at improving the efficiency of static analysis approaches that depend on solving SAT queries to detect bugs.

In approaches that attempt to improve performance of static analysis, parallelism in the code being analyzed is exploited to achieve speedup [2,14]. These approaches bank upon additional cores to achieve speedup and are agnostic pertaining to the incremental nature of the source code. In contrast, our approach gives performance benefits with the existing resources alone.

There are a variety of static analysis tools available that target specific kinds of bugs [18,19]. Techniques based on model checking [4,6] may find more bugs accurately but are not scalable for large programs. These approaches focus mainly

on finding bugs more precisely but do not attempt to address the incremental nature of software.

Incremental analysis requires that the tool identify parts of the code that have been syntactically and semantically affected by code changes. Impact analysis identifies the impact of a change on a program [10]. It has been used in the context of testing to identify tests that need to be executed after a code change. Static analysis requires the identification of parts of code that need to be reanalyzed in the incremental setting. In our approach, we analyze only the paths that are affected by code changes in the context of the property being checked.

8 Conclusions

In this paper, we identify the problem of redundant computations performed by existing state of the art static analysis engines. The redundancy exists due to the incremental nature of software development. We leverage this characteristic of software development to build a scalable static analysis that stores and uses previous analysis results effectively. We have implemented a tool, `iSATURN`, on top of `SATURN`, a state of the art static analysis engine. Our experimental results on large codebases (upto 87 KLoC) show that `iSATURN` reduces the overall analysis time upto 32%.

Acknowledgements. We thank Alex Aiken and Isig Dillig for answering our questions related to `SATURN`. We also thank Ananth Grama, Mehmet Koyuturk and Deepak D'Souza for providing useful pointers.

References

1. Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the saturn project. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 43–48. ACM (2007)
2. Albarghouthi, A., Kumar, R., Nori, A., Rajamani, S.: Parallelizing top-down interprocedural analyses. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 217–228. ACM (2012)
3. Ausiello, G., Cristiano, F., Laura, L.: Syntactic isomorphism of cnf boolean formulas is graph isomorphism complete. In: Electronic Colloquium on Computational Complexity (ECCC), vol. 19, p. 122 (2012)
4. Ball, T., Rajamani, S.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
5. Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: using static analysis to find bugs in the real world. Commun. ACM 53(2), 66–75 (2010)
6. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. International Journal on Software Tools for Technology Transfer 9(5-6), 505–525 (2007)

7. Deepakumara, J., Heys, H.M., Venkatesan, R.: Fpga implementation of md5 hash algorithm. In: Canadian Conference on Electrical and Computer Engineering, vol. 2, pp. 919–924. IEEE (2001)
8. Een, N., Sörensson, N.: Minisat: A sat solver with conflict-clause minimization. Sat, 5 (2005)
9. Hovemeyer, D., Pugh, W.: Finding more null pointer bugs, but not too many. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, pp. 9–14. ACM (2007)
10. Jashki, M.-A., Zafarani, R., Bagheri, E.: Towards a more efficient static software change impact analysis method. In: Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE 2008, pp. 84–90. ACM (2008)
11. Lauterburg, S., Sobeih, A., Marinov, D., Viswanathan, M.: Incremental state-space exploration for programs with dynamically allocated data. In: Proceedings of the 30th International Conference on Software Engineering, pp. 291–300. ACM (2008)
12. Livshits, V.B., Lam, M.S.: Tracking pointers with path and context sensitivity for bug detection in c programs. In: ACM SIGSOFT Software Engineering Notes, vol. 28, pp. 317–326. ACM (2003)
13. McPeak, S., Gros, C.-H., Ramanathan, M.K.: Scalable and incremental software bug detection. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013, pp. 554–564 (2013)
14. Mendez-Lojo, M., Mathew, A., Pingali, K.: Parallel inclusion-based points-to analysis. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2010, pp. 428–443 (2010)
15. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: Nigel Horspool, R. (ed.) CC 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)
16. Olson, M.A., Bostic, K., Seltzer, M.I.: Berkeley db. In: USENIX Annual Technical Conference, FREENIX Track, pp. 183–191 (1999)
17. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. ACM SIGPLAN Notices 47(6), 504–515 (2012)
18. Torlak, E., Chandra, S.: Effective interprocedural resource leak detection. In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, vol. 1, pp. 535–544. IEEE (2010)
19. Wassermann, G., Su, Z.: Sound and precise analysis of web applications for injection vulnerabilities. In: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2007, pp. 32–41 (2007)
20. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: POPL 2005: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 351–363 (2005)
21. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: IEEE International Conference on Software Maintenance, ICSM 2009, pp. 115–124. IEEE (2009)
22. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis, pp. 144–154. ACM (2012)