Towards Automated Bounded Model Checking of API Implementations.

Daniel Neville[1,2,3], Andrew Malton[1,4], Martin Brain[2,5], and Daniel Kroening[2,6]

[1] CHACE Centre for High Assurance Computing, Blackberry Ltd., Waterloo, Canada
[2] University of Oxford, Oxford, U.K.
[3] `daniel.neville@cs.ox.ac.uk`
[4] `amalton@blackberry.com`
[5] `martin.brain@cs.ox.ac.uk`
[6] `kroening@cs.ox.ac.uk`

Abstract

We introduce and demonstrate the viability of a novel technique for verifying that implementations of application program interfaces (APIs) are bug free. Our technique applies a new abstract interpretation to extract an underlying model of API usage, and then uses this to synthesise a set of verifiable program fragments. These fragments are evaluated using CBMC and any potentially spurious property violation is presented to a domain expert user. The user's response is then used to refine the underlying model of the API to eliminate false positives. The refinement-analysis process is repeated iteratively. We demonstrate the viability of the technique by showing how it can find an integer underflow within Google's Brotli, an underflow that has been shown to lead directly to allow remote attackers to execute arbitrary code in CVE 2016-1968.

1 Introduction

Applying automated formal methods and bug checking to application program interfaces (APIs) is a difficult and challenging process. An arbitrary set of API functions has no obvious entrance point, structure or ordering visible from the raw API code itself, despite the underlying API implementation often requiring a specific ordering of API calls to execute meaningfully. These underlying usage requirements lead to a largely manual approach being preferred by many [10], often by construction of verifiable program fragments [3]. A *fragment* is a sequence of API calls using non-deterministic input that can be verified by a formal methods engine. Because the manual construction of these fragments is time-consuming and hence expensive, our work specifically targets systems where developing a comprehensive test suite is infeasible. Manual processes are also more liable to human error.

There are many complexities to contend with when writing tests. At the highest level, the test author must ensure that the test's sequence of API calls is realistic [12]. Within this work, "realistic" refers to how a typical end-user would use the API for their own development. It is important to focus on verifying realistic code because bugs that are potentially exploitable in a normal usage pattern of the API are of arguably more interest than those that require unrealistic or pathological uses of the API. This is because there is only a finite amount of verification time available, and realistic calling sequences should be prioritised for verification. After construction of a realistic API calling sequence, the test author must ensure all variables used within the tests are instantiated meaningfully, and data structures such as buffers and pointers initialized correctly. Finally, the author has to ensure the tests are suitable for verification, and constructed in a valid input language for a verification tool.

The size and scope of APIs make them particularly difficult to test and verify, and any corresponding documentation can be complicated due to the need to be precise. Each API call may take a large number of parameters, each of which may be of variable size. Conventional test suites which aim to show an API is *completely* free of unwanted behaviour are somewhat limited due to the huge range of possible inputs. Considering a function with two 64-bit long integers as input. Each long integer can take as many as 2^{64} different values, therefore a test suite would require 2^{128} different tests just to ensure complete coverage of a single function. This is clearly infeasible.

More challengingly, even small APIs have too many permutations to allow tests to be written manually. Consider an API that has 10 API calls available, and the tester wishes to choose six API calls for a single test, in some arbitrary order. There are $10!/(10-6)! = 151,200$ permutations from these requirements, too many to write manually.

Of course, all these difficulties and more apply to any automated method for API verification or API bug checking. It should be noted that, ideally, an API should be bug free even when used outside of the authors' intended usage. Despite these challenges, it should be seen as highly desirable to have an automated method of verifying that a given API is free of common bugs such as division by zero errors, memory access outside array bounds, dereferencing of invalid pointers, memory leaks and arithmetic overflows.

Contributions.

(a) This work contributes a novel refinement-based technique for API verification (Section 3).

(b) Furthermore, given appropriate input, it contributes a method that can extract the underlying usage of the API using a novel abstract interpretation technique (Section 3.2).

(c) This work contributes a devised system then uses the extracted information to synthesise verifiable C program fragments that use the API in a realistic manner (Section 3.3).

(d) Finally, it applies bounded model checking using CBMC to verify the synthesised fragments (Section 3.4). If a false alarm is detected, the counterexample trace and underlying assumptions can be presented to the user for validation (Section 3.5), to allow novel refinement of the derived model (Section 3.6).

2 Background

2.1 Related work

Existing solutions within the sphere of API implementation verification are limited. Whilst there has been investigation into testing and verifying API implementations, these generally require comprehensive tests or verifiable programs to be provided [15]. Should a user have already spent time developing a comprehensive test suite, then any common verification framework is likely to be able to perform a reasonably effective analysis over that test suite. However, as stated previously, this work targets API implementations where developing a comprehensive test suite is either too difficult, too expensive or simply too large a project.

One previous piece of work has identified the need for gaining an understanding of the calling sequence [1], a key limitation of this approach is that it lacks a refinement feedback loop, meaning that the extracted model cannot be corrected, even if spurious counterexamples are discovered during testing. To expand upon this, our approach introduces an alternative method of extracting information via abstract interpretation, and then places this information in a more general system, one that has the ability to generate generic C programs for analysis by any analysis engine and perhaps most importantly, the ability for the user to refine their model based upon detection of false positives.

Some groups have previously used CBMC [9] for program synthesis. Di Rosa et al. [13] used CBMC to automatically generate test sets that had high branch coverage. This work is effective within the authors' domain, however it does not extend well to APIs due to the rapid state space expansion arising from API calling sequences, which in turn means only a very small subset of possible sequences can be analysed in practice. As stated previously, limited verification time means that any analysed program should be realistic and relevant, something that cannot be ensured with this technique.

Others Samak et al. [14] and Pradel et al. [11] both use test synthesis albeit for concurrent test generation. This could be a natural extension of this work.

CBMC [9] is a Bounded Model Checker for C that has the ability to verify array bounds (including buffer overflows), pointer safety, exceptions and user-specified assertions. It has very similar semantics to C; key differences include calls to declared but undefined functions are allowed, and return a non-deterministic result. Additionally, the use of declared but uninitialized variables as *rvalues* is allowed, and invokes non-determinism.

2.2 Techniques

Abstract interpretation is used for information extraction. It is a theory of sound approximation of a program [5]. In practical terms, it allows over and under-approximate conclusions to be drawn without performing all possible calculations. This balances the values of correctness and computational feasibility.

Realism spectrum. When generating verifiable fragments, it is important to generate a program which is likely meaningful under the API developers intended use case. Whilst it is not *incorrect* to generate unusual fragments like Listing 1, which may indeed turn up errors, it is relatively unlikely that a developer would intentional use the API in such a way that this will occur. It is more useful to investigate errors in *realistic* calling sequences, rather than unrealistic as these are more likely to occur in real world use.

```
api_close(x);
api_close(x);
api_close(x);
```

Listing 1: An unrealistic calling sequence.

The accuracy of a generated program as per its specification can be viewed as a spectrum. One end of the spectrum lies "complete non-determinism", where absolutely no restriction is placed upon the generated program. Anything is acceptable in a purely non-deterministic generation, but this is unlikely to reflect the API developer's intended use case and hence be a poor use of verification time.

On the other end of the spectrum lies "complete determinism", where generated programs *must* match a known calling sequence with no variation. This again is a poor approach as only witnessed input programs can be generated, which naturally cannot extend the set of verifiable programs available.

It is therefore reasonable to conclude that somewhere *within* this spectrum is the ideal place to be. Where specifically depends upon the user's intentions when using this system, although it is likely the most desirable place is somewhere near the deterministic end of the spectrum, where programs generated are similar to previously witnessed input, but sufficiently diverse such that they can exercise many different, yet realistic calling sequences.

3 Our Work

The solution has two parts: Verifiable fragment generation and model checking with refinement.

Firstly, we analyse a small sample of client code of the API. We then construct a model of realistic and typical API use based on this input, and from the model we synthesize a large number of *fragments*. These are written in standard C that can be automatically verified either with a formal verification tool that accepts non-deterministic C as input, for example CBMC.

Secondly, we apply verification to the synthesised fragments and refine the model accordingly. CBMC is used to automatically analyse the fragments according to a CBMC configuration set by the user, this allows appropriate selection of unwinding limits, etc. Should this analysis find potential bugs, an error trace is generated and presented to the user who assesses whether it is a valid and meaningful bug, or spurious. The userâĂŹs response is then used to refine the original fragments generated in Part 1. The user may direct that a derived assumption is to be discarded, modified or alternatively add a new assumption of their own if necessary. This allows flexibility, and far more accurate results than unrefined analysis.

The synthesis engine also allows a user to use an existing test system that only operates on concrete data. Support has been added for concretisation of common types. This permits the generation of fully concrete tests for use outside of formal verification.

3.1 Steps of work

Information extraction Produce an over and under-approximation of the calling sequence and calling context of the API functions given a small sample of client code of the API.

Synthesis Use the approximations to produce a synthesised fragment, which calls the API functions in the appropriate order with the right context. The fragment is exported in C. Instantiations can be non-deterministic or concrete.

Verification Using the fragment(s), attempt to verify the instrumented or developer written assertions in the API implementation using existing BMC techniques.

Validation Should a bug be detected, the counterexample trace, bug report and underlying model are presented to the user for validation.

Refinement Using the user's response, refine the abstraction and continue at **Synthesis** stage.

The key inputs to the algorithm are: a small input of client code of the API, the API header file, the API source code, a user-provided analysis location within the client code and commands for the verification engine.

The key outputs are: a model representing the calling sequence and other extracted data from the client code (to screen or serialised), synthesised verifiable program fragments and any bugs found whilst verifying the synthesised fragments.

3.2 Information Extraction

The first stage of the project is to understand the underlying usage of the API implied by the sample of client code. In particular, there is a focus on extracting likely calling sequences of the underlying API along with appropriate information regarding: the reuse of variables and other symbols between API calls, the use of constants within API calls, and how variables may be used within the body of API calls.

Data is extracted using a selection of abstract interpretation techniques. Abstract interpretation is particularly well-suited as exhaustive execution of any input files would likely be computationally infeasible, it also generates data at all locations in the program, not only the terminal states, although analysis is usually performed by extracting data at the end of the main function. Abstract interpretation allows the rapid static analysis of files, permitting the collection of necessary data in a rapid, user-friendly way.

3.2.1 Pair-wise call extraction via trace analysis

Three areas of interest are maintained to meaningfully determine the calling sequence: prefix, suffix and pair-wise ordering. Prefix refers to the first API call in a given trace, suffix refers to the final API call in a given trace and pair-wise ordering refers to how API calls are called relative to each other, e.g. `open` always immediately precedes `read`.

These are calculated over all traces in the program, with suitable merge operations used. These merge operations are namely intersection and union, and are referred to as the "Required" and "Allowed" sets respectively. In other words, required represents the domain that is valid on *all* traces through a given input and allowed represents the domain that is valid on *any* trace through a given input.

3.2.2 Prefix and suffix

The prefix and suffix represent the possible first and last API calls respectively in a given program.

Consider the program given in Listing 2. There are two paths through the program. Namely (1) `api_open_overwrite` then `api_write`, `api_close` or (2) `api_open_append`, `api_write`, `api_close`.

```
int *x;
if(non_det())
    x = api_open_overwrite("a.txt");
else
    x = api_open_append("a.txt");
api_write("Blank␣file", x);
api_close(x);
```

Listing 2: A simple API calling sequence.

There are two possibly initial API calls, `api_open_overwrite` or `api_open_append`. This generates an allowed prefix of the union of these two calls, and a required prefix of the intersection of these calls. Correspondingly, there is a single suffix call: `api_close`. Both the allowed and required suffix sets will contain this element.

The operations can be more rigorously defined as a set of operations to be applied during routine abstract interpretation (under a framework with similar semantics to the CProver Abstract Interpretation framework). Each program location retains its own set of information. For this trace analysis, the program is under-approximated to a set of connected basic blocks, each containing only a sequence of API calls, other operations are ignored.

	Allowed (union)	Required (intersection)
Domain	$SA : \mathcal{P}(A)$	$SR : \mathcal{P}(A)$
Merge	$SA_i := \bigcup_{n=1}^{\infty} SA_{i,n}$	$SA_i := \bigcap_{n=1}^{\infty} SA_{i,n}$
Transformer	$SA_o := \alpha$	$SA_o := \alpha$

Table 1: Suffix calculation operations.

	Allowed (union)	Required (intersection)
Domain	$PA : \mathcal{P}(A)$	$SR : \mathcal{P}(A)$
Merge	$PA_i := \bigcup_{n=1}^{\infty} PA_{i,n}$	$PR_i := \bigcap_{n=1}^{\infty} PR_{i,n}$
Transformer	$PA_o := \begin{cases} \alpha \cup PA_i, & \text{if } SA_o \text{ is empty} \\ PA_i, & \text{otherwise} \end{cases}$	$PR_o := \begin{cases} \alpha \cup PR_i, & \text{if } SR_o \text{ is empty} \\ PR_i, & \text{otherwise} \end{cases}$

Table 2: Prefix calculation operations.

For each state calculation, there is incoming data (prior to transform and merge) and outgoing data (after transform and merge). These sets are represented as SA, SR, PA, PR, and outgoing data is represented X_o. When a state has multiple incoming edges, the incoming data for all these edges is presented as $X_{i,n}$. This data is merged to form a single incoming data set (X_i) for calculation. States all initialise as empty.

Whenever an API call is present within an instruction, the transformer is used to calculate the outgoing set, otherwise the outgoing set is a copy of the incoming set.

The suffix domain can be calculated independently of all other information, and is therefore calculated first. The prefix domain relies upon information in the suffix domain.

Given a set of APIs A, with the transformer representing the behaviour when reaching an arbitrary API call α, Tables 3.2.2 and 3.2.2 show the operations performed.

3.2.3 Pair-wise API call ordering

Whilst knowing the first and last element within any arbitrary trace in a program is useful, it cannot be used alone to construct a realistic calling sequence.

To do this, a new domain is introduced, namely the pair-wise ordering domain. The domain is a set of pairs, where a pair ordering (x, y) represents that API call x precedes y on some trace, with no intermediate API call. For example, the allowed set $\{(\texttt{api\_open}, \texttt{api\_write}), (\texttt{api\_write}, \texttt{api\_close})\}$ represents that there is at least one trace where `api_open` precedes `api_write` and at least one trace (possibly the same) where `api_write` precedes `api_close`. Both allowed and required sets are maintained, using union and intersection accordingly to generate their data sets. The transformer uses information from the suffix data structure. Table 3 gives the operations.

3.2.4 Variable and Constant extraction

A common design pattern is to use a constant value to dictate a mode operator for an API. Consider `fopen` in Listing 3. The second input is a mode operator that dictates access mode for a given file [8]. It is unlikely that an experienced developer would use `fopen` in an arbitrary

	Allowed (union)	Required (intersection)
Domain	$OA : \mathcal{P}(A \times A)$	$OR : \mathcal{P}(A \times A)$
Merge	$OA_i := \bigcup_{n=1}^{\infty} OA_{i,n}$	$OR_i := \bigcap_{n=1}^{\infty} OR_{i,n}$
Transformer	$OA_o := OA_i \cup \bigcup_{a \in SA_i} \{(a, \alpha)\}$	$OA_o := OA_i \cap \bigcup_{a \in SR_i} \{(a, \alpha)\}$

Table 3: Pair-wise calculation operations.

manner, so using a completely non-deterministic character for *mode* is likely to be a poor choice. Instead, it should be initialised non-deterministically to one valid input within its realistic values set.

After constant propagation has been applied, the two techniques are looking over and looking into.

```
fopen (filename, mode);
```

Listing 3: `fopen`

The first technique, *looking over* is straight-forward: analysis is performed over the input files and identifying where a constant has been used as an argument in an API call. Given a function α with x arguments, a table is maintained for all arguments to track constants. This technique is also applied effectively to variables. This is vital as the same variable is frequently used in multiple API calls in a single calling sequence.

The second technique, *looking into* is more complex. Whilst examining the input data is interesting, and will no doubt generate realistic results, this is a limited approach. There may be other methods built into the code by the API developer that the API user is unaware of. This means it is unlikely that the API user will intentionally use these in an initial input file. It is therefore interesting to examine the range of possible meaningful values for an argument by looking forward, into the API code itself.

This is accomplished by performing analysis within the API bodies. Each API call is instantiated with non-deterministic arguments and constant propagation is applied. Whenever an input argument is compared against a constant value, these values can be extracted (with appropriate semantics) to generate a set of values that direct control flow. This technique results in data being extracted that allows Synthesis in different areas of the Realism Spectrum as detailed in Section 2.2.

3.3 Synthesis

The information extracted from the input data is then used to synthesise further tests for bounded model checking.

3.3.1 Procedure

Program synthesis has been implemented in CProver's Goto language framework. This language is particularly amenable to synthesis, thanks partly to its reduced instruction set and clear semantic structure. It has one additional instruction not included in C, `assume`. This instruction directs the verification engine not to analyse paths where an `assume` constraint evaluates to false.

Process.

1. Configuration initialisation and RNG seeding.
2. The *main* function is added to the synthesised program (SP.)

3. Known API prototypes are added to the SP.
4. Known symbols that will be necessary are added to the SP.
5. A walk [See: The Walk] is performed over the information extracted from the tests, specifically a walk is performed over the calling sequence information. This adds API calls to the SP. When each API call is added, the `assume` database is checked to see whether to insert an `assume` statement prior to API call.
6. The SP is exported as ANSI-C code to file.

The Walk.

1. An appropriate API function is chosen.
   (a) For the first call: A function that is required (or at least allowed, configuration dependent) to appear first.
   (b) For the last call: A function that is required (or at least allowed, configuration dependent) to appear last.
   (c) For any other call: A function which is required to appear following the previous call (or at least allowed, configuration dependent) or another function dictated by a configuration strategy such as randomisation.
2. The API's arguments types are extracted and evaluated, including return type.
3. Symbols are selected with appropriate instantiation to support the API call [See Symbol Selection.]
4. Instantiated pointers are pointed to appropriate data structures [with necessary recursive instantiation].

Symbol Selection. Given an arbitrary API call with n arguments and a return value k, it is essential to ensure appropriate symbols are used to ensure data is passed through the API calling sequence in a realistic method.

Symbol selection supports several different techniques. It is heavily configuration dependent to ensure good quality results.

Given API call α at position k

(a) If a suitable constant (string, numeric, etc.) is used at $\alpha : k$, add this to the consideration list.

(b) If a suitable variable is used at $\alpha : k$ that has been used previously in an API call, and there is suggestion of such a pattern in the input data: add this to the consideration list.

(c) If there is an appropriate constant or symbol available: select according to some strategy.

(d) Otherwise: Declare and instantiate a new variable in a suitably non-deterministic manner (unless full concretisation is enabled in configuration.).

Instantiation. Instantiation is non-trivial, especially for non-primitives. Consideration must be made as to how to initialise pointers, `structs` and other data structures. Because CBMC will be used to verify the synthesised programs, it is not necessary to assign a concrete value to all declared variables. For example, given the code in Listing 4, CBMC is perfectly capable of handling the inherent non-determinism.

```
int x;
int y;
int z = api_do(x, y);
```

Listing 4: A simple, valid calling sequence

However, for more complex structures such as pointers, failing to instantiate will result in trivial errors occurring, such as null pointer exceptions. This is demonstrated in Listing 5. On line 3, a deference takes place, but CBMC cannot assume that the pointers are instantiated correctly, therefore any instrumented pointer assertions will return `false`. To handle this, all pointers must be instantiated (or intentionally left null, as per configuration) before use.

Whenever a new pointer symbol is created for the purpose of being used in an API, a corresponding symbol must be created and instantiated for this new pointer symbol to point. This is demonstrated for integers in Listing 6. This will ensure both variables are suitable instantiated prior to call. This technique is also used for pointers to large objects such as buffers.

This strategy can be applied recursively for non-primitives. Where a necessary `struct` or similar is declared and then each internal primitive is initialised with the above strategy. A challenge arises when a `struct` contains a pointer to itself, as this can lead to an infinite initialisation loop, this behaviour is handled by modifying the Synthesis Engine's configuration.

```
int *x;
int *y;
int z = api_do(*x, *y);
```

Listing 5: Example of use of non-initialised pointers

3.4 Verification

Once the fragments have been generated, the user's desired testing or verification procedure is followed. The intended verification engine for this project is CProver's CBMC. The generated fragments are compiled into CProver's intermediate reduced instruction-set language (GOTO), and bug checks are instrumented into the program in the form of assertions. CBMC then attempts verification on each program under the configuration provided by the user.

For each program, if an assertion is violated, then a counterexample trace is generated for validation. If no assertion is violated then CBMC will proceed to the next program.

```
int *x;
int x_pointed_to_1;
x = &x_pointed_to_1;
int *y;
int y_pointed_to_1;
y = &y_pointed_to_1;
int z = api_do(*x, *y)
```

Listing 6: Initialisation of pointers.

3.5 Validation

If a counterexample *is* found within a fragment, it is presented to the user. The user can then manually inspect the calling sequence that lead to the bug.

If the user confirms the bug is genuine, the violated property is identified and the corresponding line of raw source code logged accordingly. The user can then review this later to ensure bugs in the underlying are fixed. Once the user believes they have removed the bug, they can re-run the same synthesised fragment to give a better indication of whether the bug is fixed. The user can then choose to continue verification or stop to fix any bugs found.

If however, the bug is not genuine, there may be a miscalculation in the underlying API calling model. Therefore, the user may wish to *refine* the model.

3.6 Refinement

To refine the model, the user is provided with the property that was violated, a counterexample that led to the bug and the underlying assumptions that were generated in the Information Extract stage of the process.

The user can at this point decide on the most appropriate action. They can ignore the error, and continue execution. They can abandon execution, and find an alternative input, or most preferably, they can refine the underlying model. The model's constraints can be added to, modified, or deleted. This process can be repeated until the user is satisfied they have removed the offending incorrect rules. After the user has refined the model, the process returns to Step 2: Synthesis. New fragments are generated using the refined model, and analysis continues.

The refinement approach borrows from the underlying principles behind counterexample guided abstraction refinement (CEGAR) [4], a specific similarity occurs when a failing proof is found, both CEGAR and this approach result in analysis of the cause of failure, specifically the counterexample trace, refining this model allows a more precise verifiable program fragment to be synthesised, and hence more effective analysis.

4 Results

To show the viability of this novel refinement-based technique a test candidate was selected from Github. Brotli is a generic-purpose lossless compression algorithm used for data compression [2]. Both an encoder and decoder are available. Specifically, the decoder was chosen for analysis.

The decoder is attractive for this type of analysis for several reasons. Firstly, Brotli's decoder is written in pure C with few external dependencies. This means its build process can easily be modified to use CProver's compiler. Thirdly, there was at least one known bug in Brotli (CVE-2016-1968 [6]) suggesting that effective verification techniques had not been applied to Brotli, this was eventually fixed by the developers in commit 37a320d [7]. Finally, the code was large and complex enough to perform meaningful and non-trivial evaluation. Brotli's decoder has 14,000 effective lines of C code, of which 2,000 are contained in C header files. Analysis was performed using an Ubuntu 64-bit virtual machine running in Oracle VM VirtualBox with 8 GB of RAM and with non-dedicated access to two cores of an Intel 4810MQ at 2.80 GHz.

Once Brotli had been configured to compile under CBMC, the code was analysed and a realistic test was written using components from the Brotli API; manual creation was necessary due to the limited test data within Brotli. This test was executed concretely, but produced no bugs or assertion violations. This sample test was then used as input for the Data Extraction Engine and Synthesis Engine, which created multiple fragments that could be used under a verification engine. These were then analysed using CBMC's signed overflow and bound checks. The CVE vulnerable property was placed under scrutiny. CBMC found a possible arithmetic underflow for the property directly associated with the exploit. The violated assertion directly corresponds to the patch added by the authors to mitigate this bug.

The performance of information extraction was rapid, even within a virtual machine environment, Step One: Information Extraction was complete within 0.4 s. 1,000 verifiable Brotli fragments could be synthesised from this data in 6s. The model was refined once to modify an instantiation.

5 Conclusions and Future Work

We have shown in this work that API analysis, despite its challenges, can be confronted using a mix of formal verification, data mining techniques and program synthesis. We have shown that underlying API usage data can be extracted rapidly, and used to synthesise verifiable fragments.

We have demonstrated a method to allow refinement of the fragments after verification to improve the model, and explained how the user can interact to improve the quality of their results quickly, without being required to manually author numerous tests themselves. Finally, we have shown that it can be applied effectively to real-world code.

To expand upon this, we have devised a new abstract interpretation technique for under-approximate information extraction using regular expressions which is being implemented, as well as the introduction of other domains (e.g. interval domain) to better represent some APIs. We are also intent on developing far richer results to show the effectiveness of the technique. Additionally, we desire the introduction of a formal grammar for synthesis. Finally, to encourage use of the technique, a user interface will be created to connect the currently distinct components and it is hoped this work will become part of the open-source CProver suite of tools.

References

[1] Mithun Puthige Acharya. *Mining API Specifications from Source Code for Improving Software Reliability.* PhD thesis, North Carolina State University, 2009.

[2] J. Alakuijala and Z. Szabadka. Brotli compressed data format. `http://www.ietf.org/id/draft-alakuijala-brotli-09.txt`, April 2016. [Online; accessed 17-April-2016].

[3] Michael Churchman. API testing: Why it matters, and how to do it. `https://blog.udemy.com/api-testing/`, April 2014. [Online; accessed 17-April-2016].

[4] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings*, chapter Counterexample-Guided Abstraction Refinement, pages 154–169. Springer, Berlin, Heidelberg, 2000.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252, 1977.

[6] CVE. CVE-2016-1968. Available from MITRE, CVE-ID CVE-2016-1968., January 20 2016.

[7] Google. Brotli commit 37a320d. `https://github.com/google/brotli/commit/37a320dd81db8d546cd24a45b4c61d87b45dcade`, 2016.

[8] The IEEE and The Open Group. The open group base specifications issue 7: fopen. `http://pubs.opengroup.org/onlinepubs/9699919799/functions/fopen.html`, 2013. [Online; accessed 17-April-2016].

[9] Daniel Kroening and Michael Tautschnig. CBMC C Bounded Model Checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413, Berlin, Heidelberg, 2014. Springer.

[10] Josh Poley. Best practices: API testing. `https://msdn.microsoft.com/en-us/library/cc300143.aspx`, February 2008. [Online; accessed 17-April-2016].

[11] Michael Pradel and Thomas R. Gross. Fully automatic and precise detection of thread safety violations. *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation - PLDI '12*, page 10, 2012.

[12] Asha K. R. and Shwetha D. J. API testing: Picking the right strategy. In *Pacific Northwest Software Quality Conference 2015*, pages 261–270, 2015.

[13] Emanuele Di Rosa, Enrico Giunchiglia, Massimo Narizzano, Gabriele Palma, and Alessandra Puddu. Automatic generation of high quality test sets via CBMC. In Markus Aderhold, Serge Autexier, and Heiko Mantel, editors, *VERIFY-2010. 6th International Verification Workshop*, volume 3 of *EPiC Series in Computing*, pages 65–78. EasyChair, 2012.

[14] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2015*, pages 175–185, 2015.

[15] Charles P. Shelton, Philip Koopman, and Kobey Devale. Robustness testing of the Microsoft Win32 API. In *2000 International Conference on Dependable Systems and Networks (DSN 2000)*, pages 261–270. IEEE, 2000.