Formal Verification of Secure Software: Testing the Limits of Facebook's Quandary

The University of Melbourne

Skye R. McLeman (809770)

Master of Information Technology (Computing)

COMP90055 Research Project (25 pts)

Type of Project: Conventional research project

Semester 2, 2018

Supervisors: Dr Toby Murray & Dr Gidon Ernst

Final Report

CONTENTS

Abstract

This report examines Quandary, an experimental module for static taint analysis found in Facebook's open-source static analysis tool, Infer[1]. Quandary is designed to help assist software developers in effectively identifying and fixing potentially unsafe information flows in large applications written in the C, C++, Objective-C, or Java programming languages. One of the main contributions of this research is a suite of test cases which demonstrate some of the specific capabilities and limitations Quandary possesses when it comes to analyzing code written in the C programming language. We document these capabilities alongside our intuitions and expectations as to how the program would behave if it were to effectively flag potentially unsafe information flows to the developer. For example, although Quandary is capable of tracking tainted data written in C code in which a sanitizing function is called from within another function, Quandary fails to flag a taint error if tainted data is copied back into a buffer after it has been sanitized. In addition to this, we also provide some background information on formal verification and software security in general, and discuss why this discipline is critical to the success of modern highly secure software engineering. Finally, we outline and discuss multiple potential avenues for further research.

I. INTRODUCTION

As software projects become orders of magnitude larger, development —- specifically, *secure* software development — is becoming increasingly difficult. For applications consisting of millions of lines of source code (see [1] for a good visualisation of this), expecting software engineers to simultaneously smash out code for rapid release and maintain its high integrity is an impossible ask. While unit and regression testing can help up to a point, there are certain classes of bugs and security vulnerabilities which can only (or at the very least, are much more likely to) be caught with the assistance of automated tools such as static program analyzers and taint detectors.

Facebook's Infer is one such static analysis tool, and Infer is unique in that it is highly scalable, suitable for continuous development and deployment, and compatible with a rapid, iterative software development lifecycle [2] [10]. In the past, one of the barriers to wider adoption of static analysis has been the fact that the tools at the time required an entire code-base to be analyzed all at once, and given the speed of those tools, this generally meant running an analysis overnight, i.e. only once a day. Infer has no such limitations: due to the compositionality of its design, separate modules of a code-base can be analysed at different times, cached, and the results combined to provide assurances that cover the entire system. Infer's speed, combined with this property of compositionality, means that Facebook has managed to successfully incorporate it into its developers' every-day build processes and pipelines; for example, each time a developer submits a pull request, the code they are working on is analysed

[1]https://fbinfer.com. We used version 0.14.0

by Infer, and comments are automatically inserted into the source code which indicate to developers potential bugs or issues which can then be addressed before any commits are merged into the master branch of the project.

In terms of the theory which it is built upon, Infer leverages recent advances in Hoare logic, abstract interpretation, separation logic, and bi-abduction [3]. We will not discuss the theoretical underpinnings behind Infer or Quandary in any detail here, but if the reader is interested in such information they are encouraged to read [20], [21], and [22]. Nor will we discuss in any detail Infer's source code implementation, which is written in OCaml and freely available via GitHub[2].

In this paper we examine the behaviour of Infer's experimental module Quandary, which attempts to identify unsafe information flows using static taint analysis. We will look at several test cases developed throughout the course of this research which serve to highlight a small selection of Quandary's capabilities and limitations when analyzing code written in the C programming language. In this research we had three main objectives and associated outcomes, listed here in order of priority and significance (in terms of contribution):

1) document Quandary's behaviour—in particular, some of its capabilities and limitations—and create a suite of open-source test cases which demonstrate this behaviour;
2) provide some background information and context on formal verification and its motivation in general, and explain why this field is crucial to the success of modern highly secure software engineering;
3) provide some suggestions as to potential avenues for further research.

II. Formal Verification

A. *What is Formal Verification?*

Formal verification is a technique whereby we abstract away certain implementation details to create a mathematical or logical model of a particular system—for example, a CPU architecture such as Intel x86 or ARM—which we can then reason about systematically. In other words, we *formalize* the properties and behaviour of a system under consideration—usually using some form of mathematical or symbolic logic[3]—then reason about the limits of this behaviour precisely with the use of that formalism.

In fact, we can use logic to create a formal *proof* (sometimes with the aid of an automated *proof-assistant*[4]) that certain properties of this system will hold under certain conditions, and that entire *classes*

[2]https://github.com/facebook/infer

[3]Sometimes we will create entirely new logic systems in the process, if necessary, in order to accomplish this.

[4]See e.g. [23], [24], and [25] for some useful background information on and examples of proof assistants.

of bug or exploit are impossible. For example, it is possible to prove that a particular C program is invulnerable to buffer overflow attacks. Similarly, it is possible to prove that a particular web-application is not at all vulnerable to SQL-injection [15]. Given the fact that command-injection attacks have consistently made it to first place in the OWASP Top 10 Application Security Risks each year from at least 2013 to 2017 [29], the idea that we can construct a formal proof to show that a program is completely immune to this entire class of attack is a tantalising one indeed, and certainly no mean feat. Imagine being able to sit back, relax, and bask in the glorious knowledge that you have a *mathematical proof* which demonstrates the invincibility of your code!. Fortunately, it turns out that this idea can, in fact, be made manifest in reality—with a significant set of substantial caveats, of course.

B. What Are the Problems With Formal Verification?

At the end of the day, the usefulness and success of formal verification as a tool depends largely upon how well all of our abstract models map to reality. We must always be asking ourselves the following question: is it possible that this model will fail to accurately account for some other (possibly not-yet-known) class of attack and its associated behaviours?

Some illuminating examples of how model-reality mismatches can occur can be found in the many recent vulnerability discoveries and disclosures relating to *side-channel* or *timing* attacks. Murray discusses these at length in [4], and explores some of the issues inherent in modelling and formalising systems in general, using recent exploits such as Rowhammer [11], Spectre [13], and Meltdown [14] as concrete examples. As vague as 'abstracting away elements in order to model a system' may sound to someone unfamiliar with the field, errors in this process have serious real-world ramifications: c.f. the Spectre attack which takes advantage of a vulnerability in the way some Intel processors implement a feature known as *speculative execution*. It would be possible to have a formal proof or series of proofs that these processors are immune to a whole range of different exploits and attacks, but if our model fails to take into account side-channels such as low-level caches, then the usefulness (and ultimate meaning) of these proofs is going to be brought into question.

While it is certainly true that a proof of a program's invulnerability to a particular attack class is, by definition, irrefutable (otherwise it wouldn't be a 'proof'), and is in every instance a testament to the power of formal verification, it is entirely possible that the abstraction used fails to model an aspect of the system which has significant security implications: this is what tends to happen in reality, with the consequence that any security guarantees given must be very carefully qualified.

C. Why is Formal Verification Useful?

We proceed under the assumption that formally verifying security-related properties is a useful endeavour. Murray outlines multiple reasons why this (usually) difficult and expensive exercise can be beneficial [4]:

1) Proofs can provide qualified guarantees as to the security of a piece of software (with the caveat that the formal model accurately reflects reality): In other words, formal verification can help us to increase our confidence in the security of our software. At the very least, by having the capacity to eliminate certain classes of attack from our risk assessments, proofs can allow us to use our resources more effectively, since we can focus only on attacks which we know the system to be vulnerable to.

2) Proofs 'force careful and rigorous understanding of a system (i.e., system model) and its operation': Creating a sound formal proof of a system cannot be accomplished without gaining an extremely deep understanding of a system in the process. Thus there is inherent value in the process of attempting to complete a formal proof for a property of a system, regardless or whether we are able to produce a sound proof at the end: the very act of attempting the proof adds immensely to our knowledge of the system, which can only be a good thing.

3) Proofs have economic value: For example, a product formally-verified as secure can be marketed as such, and used to distinguish it from competing applications; not only that, a company that engineers software formally verified as secure could also incorporate this fact into its branding and use it to distinguish it from its competitors. A vendor could also use the feature of Formally-Verified Secure® as an excuse to hike up the price of such products. Additionally, there is the potential for formal verification to help limit the commercial liability of companies and protect against some classes of legal issue; for example, it might be useful for a corporation to be able to boast that it purchased only software that was formally verified as 'secure' if they were liable for a serious security breach involving customer or supply-chain data.

D. Do Any Practical Products of Formal Verification Already Exist?

Yes, in fact, many recent advances have been made in the field of formal verification of software security [16]; for example, the provably-secure sEL4 Linux Mikrokernel was created years ago and is being used in Boeing's AH6 'Unmanned Little Bird (ULB)' aircraft.

Also, in addition to aiding in the development of new highly secure software, formal verification can be used to verify the security or functional correctness of systems which have already been created; for example, there are now formal proofs for the security of protocols such as TLS (see e.g. [26], [27]).

This is but a tiny fraction of some of the recent advances in formal verification; there are many more. Indeed, it is remarkable that the field continues to make such rapid progress, especially considering the fact that the spotlight is (at present) firmly focused on other areas, such as artificial intelligence and machine learning, for example. It is interesting to note that for all of the media attention generated by the multitude of recent high-profile data-breaches and security incidents, very little is known about and hardly any attention is given to one of the most effective solutions — a solution which actually manages to address the underlying issues at a fundamental level.

Despite the flurry of recent advances, however, formal verification still largely remains an obscure and arcane concoction of art and science; pursued only by persistent mystics who are able to focus intently on the logical sigils long enough to achieve a state of security gnosis. Of course, there are many factors involved; for example, as mentioned earlier, formal verification tools used to have a reputation (and still do, to some extent) for being too expensive or slow for use in practical, day-to-day software development. However, times have changed: long gone are the days when formal verification was too expensive or slow for real-world applications. Thanks to sophisticated tools such Quandary and Infer, we now live in a world of fast, cheap, scaleable, and (perhaps more importantly) *accessible* automated quality control. The question is, how well do the models these tools rely on stack up to reality?

III. PRINCIPLE OF NONINTERFERENCE

A. *Definition*

A key concept in formally verifying software security is that of *noninterference*. Broadly speaking, noninterference refers to the property of a secure program whereby the data in *secret* (or *private*) variables has no effect on (i.e. is not able to *interfere with*) the contents or output of any *non-secret* (i.e. *public*) variables. Secret data can still be manipulated internally by the program, but only so long as it does not influence the program's public output in such a way that its secrets can be revealed to an attacker.

There are multiple different variations on the property of noninterference, many of which have been precisely formalised and defined. For example, Hedin and Sabelfeld [5] provide a comprehensive overview and taxonomy of several different information-flow control policies and types of noninterference, drawing on research which includes Denning's pivotal work from the 70s [6].

B. *Types of Noninterference*

Hedin and Sabelfeld identify and characterise four main types of noninterference:

1) termination-insensitive noninterference (TINI);

2) termination-sensitive noninterference (TSNI);
3) progress-insensitive noninterference (PINI); and
4) progress-sensitive noninterference.

Effectively, these categorisations pertain to whether or not a program can leak sensitive information to an attacker who is able to observe if it has terminated or diverged (*termination-sensitivity*); and whether or not a program can leak sensitive information to an attacker through the attacker's observation of intermediate steps of the computation (*progress-sensitivity*).

1) Termination-sensitive noninterference: termination-sensitive noninterference relates to a program's vulnerability to leaking sensitive information to an attacker who is able to observe whether it has terminated or diverged [18].

A simple example of termination-sensitive noninterference:

```
    if (public_value == private_value) then while (true) do skip done
```

Listing 1. A simple example of termination-sensitive noninterference

In the example above, an observer can deduce whether public and private values are the same based on the termination behavior of the program. So long as **public_value** is equal to **private_value**, the program remains in an endless loop, which means an attacker who is able to observe that the program is stuck in an endless loop can infer the secret value.

While it is possible to prove that a program is not vulnerable to exploits relating to termination-sensitivity, it is generally harder to do so for than the weaker definition of security which still allows programs to leak sensitive information via their termination behaviour. The degree to which this affects the real-world security of an application is an open question: how much more secure is a program which satisfies the property of progress-sensitive noninterference than one which has only the property of process-insensitive noninterference? Is progress-insensitive noninterference enough? Or is it pointless to even attempt to secure a program when it is able to leak sensitive information via its termination behaviour? These are the kinds of questions which need to be asked, and the quicker we can answer them, the faster we will be on our way toward providing meaningful assurances as to the security of a given piece of software.

2) Progress-sensitive noninterference: Progress-sensitive noninterference is concerned with whether or not a program can leak sensitive information through to an attacker who is capable of observing the intermediate steps of the computation [17].

A basic example of progress-sensitive noninterference (taken from [17]):

```
for (i = 0; i < MAXINT; i++) {
    while (i == secret) do skip;
    output(0);
}
```

Listing 2. A basic example of progress-sensitive noninterference

This example illustrates how a program can leak sensitive information through to an attacker who is able to observe its *progress* (i.e. intermediate output). The program will output the character '0' up until the public counter **i** is equal to the private variable **secret**, at which point it will begin to loop indefinitely. An attacker who can observe this is thus able to infer the value of a private variable (in this case **secret**) simply by examining the output of the program during the intermediate steps of its computation. Again, what a property such as progress-sensitive noninterference actually means for software in the real world is an as-yet unanswered question, and the reader is encouraged to consult [4] for a recent and thorough exploration of these types of issues.

IV. STATIC TAINT ANALYSIS

A. What is Static Taint Analysis?

At a high level, taint analysis aims to trace the flow of *tainted* (i.e. potentially malicious) input through a program, and alert or warn the developer of any potentially unsafe information flows. Unsafe information flows involve tainted data passing from a *source* to a *sink* without getting *sanitized* (i.e. having any malicious data removed) at some point along the way. Taint analysis can be performed *statically* (i.e. on dormant source code), or *dynamically* (i.e. at run-time).

Static taint analysis, and taint analysis in general, are by no means new concepts or ideas. Taint analysis has been around since at least 1989 when it became a feature of the Perl programming language [28], continues to be a commonly-used feature of the Ruby programming language, and is considered an integral part of many modern software development pipelines.

In static taint analysis, the developer must define *sources* of potentially tainted data; for example, one might define a function such as **gets** as a source in a command-line program which takes input from the user. This means that anything the end-user inputs into the program is considered tainted (i.e. potentially malicious).

Similarly, *sinks* (i.e. outputs) must also be defined by the developer: for example, a developer might define the function **puts** in a command-line program as a sink. If any tainted data managed to make its way from a source to a sink, then our analyzer should flag some kind of error—for example, Quandary produces a 'taint error', warning the user and indicating on which line it thinks the violation has occurred on in the source code:

```
Found 1 issue

correctly-recognises-sanitized.c:14: error: QUANDARY_TAINT_ERROR
  Other(gets()) at line 12, column 5 ~> Other(puts()) at line 14,
     column 5.
  12.         gets(buf);
  13.         //buf = check(buf);
  14. >       puts(buf);
  15.         free(buf);
  16.     }


Summary of the reports

  QUANDARY_TAINT_ERROR: 1
```

Listing 3. Example of Quandary's taint error

So what use is taint analysis if evidently any basic program which takes user input and produces output is going to get flagged with a taint error? Well, as it turns out, there is a way to express the notion that we have 'cleansed' or *sanitized* the offending input, in which case we can allow it to reach a sink and become part of the program's output. A **sanitizing** function is responsible for removing any potentially malicious input; for example, strings of characters implementing attacks such as SQL-injection or Cross Site Scripting (XSS) might be stripped from the input. In Perl this is generally accomplished with the use of regular expressions. In Quandary, we are free to implement sanitizers however we like in the C programming language; indeed, as with Perl's built-in taint checker, Quandary does not care about the implementation or the correctness of the sanitizer at all: Quandary's only concern is that *some* sanitizing function is being called on some potentially malicious input—how that sanitization occurs is completely up to the developer.

Here is an example of some C programming language source code which demonstrates Quandary's ability to flag taint errors:

```
#include <stdio.h>

// sanitizing function
char *check(char *b) {
    return b;
}

void f() {
    char *buf = malloc(100);
    if (buf == 0) return;

    gets(buf);
    buf = check(buf);
    puts(buf);
    free(buf);
}
```

Output:

```
# infer --quandary run -- clang -c simple-taint-analysis.c -o simple-
    taint-analysis.o
Capturing in make/cc mode...
Found 1 source file to analyze in /opt/infer-linux64-v0.14.0/examples/
    infer-out
Starting analysis...

legend:
  "F" analyzing a file
  "." analyzing a procedure

F...
  No issues found
```

In the above code we simply allocate a 100-byte buffer called **buf**, get input from the user via the terminal and store it in **buf**, call a sanitizing function on **buf** called **check**, then finally, we output the contents of **buf** to a sink, **puts**. Even though we are dealing with tainted data, because the data is sanitized before reaching a sink, Quandary does not flag a taint error here which is what one would expect.

It is worth pointing out that the sanitizing function here, **check**, does nothing except return the string which is passed to it as an actual parameter. Quandary is not concerned with the specifics of how sanitizing functions are implemented, or whether they actually manage to sanitize the data at all – all Quandary cares about is that *some* sanitizing function is called on the tainted data before it reaches a sink.

B. What is the Relationship Between Static Taint Analysis and Noninterference?

Although they are two distinct concepts, static taint analysis is commonly used to approximate reasoning about noninterference. For a detailed discussion of this, please see Schoepe, Balliu, Pierce, and Sabelfeld's 'Explicit Secrecy: A Policy for Taint Tracking' [9].

V. METHODOLOGY

A. Operating Environment

All tests were conducted using version 0.14.0 of Infer using the pre-built binary provided by Facebook. It is worth noting that Facebook are currently up to version 0.15.0 and it is entirely possible that the behaviour documented in this research is no longer up-to-date.

The test environment consisted of a cleanly-installed Fedora 27 virtual machine running on VirtualBox as a hypervisor. Although the binary did not work straight off-the-bat, this was easily remedied by installing some missing library headers, and we were soon able to begin investigating its behaviour. Later on, we also re-ran all the tests using the Docker image which Facebook provides, and were interested to find that some of the results had changed (i.e. some behaviours had reversed and Quandary was now flagging code with taint errors where it hadn't before, and vice versa).

B. Creating the Test Suite

We went about investigating Quandary and building the test suite in a very ad-hoc fashion: we began with some simple example code which illustrated a few of Quandary's capabilities, and from that point on began to regularly experiment and create other examples, usually adding various constraints (e.g. focusing on pointer manipulation) each time.

VI. Quandary's Behaviour

Facebook use a static analysis program called Infer as part of the day-to-day build pipelines for both their Android and iOS apps, including the main Facebook apps, Facebook Messenger, and Instagram, amongst others [10]. Infer contains an 'experimental' (i.e. work-in-progress) module called Quandary, which can attempt to perform static taint analysis for programs written in the C, C++, Objective-C, or Java programming languages.

Throughout the course of this research we created a set of test cases which illustrate some of Quandary's capabilities and deficiencies when it comes to taint-tracking code written in the C programming language. In doing so we have helped to document the behaviour of a useful tool for modern secure software development and hope to motivate further research into analysing, extending, and maintaining it. We also hope to give the reader a sense of some of the practical applications (and limitations) of formal verification in general.

A. *Using* ***.inferconfig*** *to Configure Sources, Sinks and Sanitizers*

1) Overview: Sources, sinks, and sanitizers for static taint analysis are defined for Quandary in the **.inferconfig** configuration file. This configuration file is written in the familiar JSON format, i.e. it is a hierarchy of nested objects, attributes, and arrays.

Here is the configuration we used for all of our test cases during the course of this project:

2) .inferconfig Configuration Used in Test Cases:

```
{
    "quandary-sources": [
                    { "procedure": "gets", "kind": "PrivateData", "index":
                        "0" },
                    { "procedure": "gets_call_sink_within_source", "kind":
                        "PrivateData", "index": "0" },
                    { "procedure": "gets_different_kind", "kind": "98
                      j32fj2938fj", "index": "0" },
                    { "procedure": "gets_invalid_index", "kind": "whatever
                      ", "index": "1" },
                    { "procedure": "gets_second_index_source", "kind": "
                      88888", "index": "1" },
                    { "procedure": "gets_second_index_sink", "kind": "
                      fddfdf88888", "index": "0" }
```

```
        ],
    "quandary-sinks":    [
                { "procedure": "puts", "kind": "PrivateData" },
                { "procedure": "puts_call_sink_within_source", "kind":
                    "PrivateData" },
                { "procedure": "puts_different_kind", "kind": "
                   JOIJ838j3f838" },
                { "procedure": "puts_invalid_index", "kind": "
                   whateverX" },
                { "procedure": "puts_second_index_source", "kind": "
                   77777" },
                { "procedure": "puts_second_index_sink", "kind": "
                   wywyw77777", "index": "1" }
        ],
    "quandary-sanitizers": [
                { "procedure": "check", "kind": "PrivateData" }
        ]
}
```

The configuration file is relatively self-explanatory. There is a section for defining each of our various sources, sinks, and sanitizers, and each of these sections contains an array of objects (each of which has the key ”procedure”) which define which functions in the program belong to which category. For example, here we have defined the functions **gets** and **gets_invalid_index** as sources, and the functions **puts** and **puts_invalid_index** as sinks. We have only defined a single sanitizer function: **check**.

There are three main attributes which can be set for each definition of a source, sink, or sanitizer in the **.inferconfig** file:

1) **”procedure”**: the name of the function which will be assigned to the category this definition is nested under (one of **”quandary-sources”**, **”quandary-sinks”**, or **”quandary-sanitizers”**).
2) **”kind”**: this appears to be an arbitrary text field which the developer can use to document their definitions; please see the section below discussing this attribute for more information.
3) **”index”**: allows the developer to specify the index (starting at 0) of the formal parameter of the function which Quandary will attempt to track taint for. For example, we have set the index attribute of our entry for the **gets** function to 0, which means we are interested in having Quandary

track taint for input which arrives at our program as the first actual parameter of this function.

3) The "kind" Attribute: We were surprised to discover that the **kind** attribute of each definition appears to have no effect on Quandary's behaviour at all. As is evident in the configuration file above and the source code below, we tested this by trying various combinations of strings for the "kind" attributes of several sources and sinks. For example, our source **gets_different_kind** has a "kind" attribute of "98j32fj2938fj", while our sink **puts_different_kind** has a "kind" attribute of "JOIJ838j3f838". It did not seem to matter which string we used for the "kind" attribute of any combination of sources or sinks; none of this seemed to have any impact on the outcome of any of Quandary's analyses. The "kind" attribute appears to simply be an arbitrary string which the author assumes is intended to aid the developer in documenting their configuration.

```
#include <stdio.h>

char *check(char *b) {
    return b;
}

char *gets_different_kind(char *str) {
        return gets(str);
}

int *puts_different_kind(const char *str) {
        return puts(str);
}

void f() {
    char *buf = malloc(100);
    if (buf == 0) return;

        gets_different_kind(buf);
        buf = check(buf);
        puts_different_kind(buf);
        free(buf);
}
```

Listing 4. C source code for tests using source and sink with different "kind" attribute in .inferconfig. Both the source and sink were given a random string as the "kind", and this does not appear to affect Quandary's behaviour, i.e. these attributes appear to simply be labels for the convenience of the user.

B. SOME EXAMPLES OF QUANDARY'S CAPABILITIES

1) Quandary correctly recognizes when tainted data is sanitized before reaching a sink:

```
#include <stdio.h>

// sanitizing function
char *check(char *b) {
    return b;
}

void f() {
    char *buf = malloc(100);
    if (buf == 0) return;

    gets(buf);
    buf = check(buf);
    puts(buf);
    free(buf);
}
```

Output:

```
...
  No issues found
```

Listing 5. Quandary detects sanitizing of tainted data

In this test case we simply obtain input from a tainted source (**gets**), run it through a sanitizer function, then output it to a sink (**puts**). Quandary reports 'No issues found', which is exactly what we would expect and want to happen.

Note, however, that if we analyze the same code with the call to the sanitizer commented-out, we get the following result instead:

```
# infer --quandary -- clang -c correctly-recognises-sanitized.c
Capturing in make/cc mode...
Found 1 source file to analyze in /opt/infer-linux64-v0.14.0/examples/
    infer-out
Starting analysis...

legend:
  "F" analyzing a file
  "." analyzing a procedure

F..
Found 1 issue

correctly-recognises-sanitized.c:14: error: QUANDARY_TAINT_ERROR
  Other(gets()) at line 12, column 5 ~> Other(puts()) at line 14,
     column 5.
  12.         gets(buf);
  13.         //buf = check(buf);
  14. >       puts(buf);
  15.         free(buf);
  16.    }


Summary of the reports

  QUANDARY_TAINT_ERROR: 1
```

Listing 6. Quandary correctly flags taint error where data is not sanitized

Here we can see Quandary correctly reports that there is a taint error in the code, since the tainted data is able to make it from source to sink without being sanitized at some point. Again, this is exactly how we would expect and want the program to behave.

2) Quandary correctly allows sanitizer to be called from another function:

```
#include <stdio.h>

// sanitizing function
char *check(char *b) {
    return b;
}

char *call_sanitizer(char *b) {
    return check(b);
}

void f() {
    char *buf = malloc(100);
    if (buf == 0) return;

    gets(buf);
    buf = call_sanitizer(buf);
    puts(buf);
    free(buf);
}
```

Listing 7. Quandary correctly allows sanitizer to be called from another function.

Output:

```
No issues found
```

As expected, Quandary does not flag a taint error here. This would seem to fit with our intuition, as the sanitizer is simply being called from within another function. The important thing here is that the sanitizing function is being called on the tainted data before it reaches a sink – whether the sanitizing function is called from within another top-level function, or is a top-level function itself, is not relevant.

3) Quandary is able to correctly analyze code nested within C pre-processor conditional statements:

```
#include <stdio.h>
```

```
char *check(char *b) {
    return b;
}
void f() {
    char *buf = malloc(100);
    if (buf == 0) return;

    gets(buf);
#if 0
    buf = check(buf);
#endif
    puts(buf);
    free(buf);
}
```

Listing 8. Quandary fails to correctly analyze code within C pre-processor conditionals

Output:

```
Found 1 issue

preprocessor-conditionals-if-0.c:14: error: QUANDARY_TAINT_ERROR
  Other(gets()) at line 10, column 5 ~> Other(puts()) at line 14,
     column 5.
  12.         buf = check(buf);
  13.     #endif
  14. >       puts(buf);
  15.         free(buf);
  16.     }
```

This code is correctly interpreted by Quandary and produces the behaviour one would expect and desire. Since the call to the sanitizing function **check** is nested within a C pre-processor conditional statement which always evaluates to false, the sanitizing function is never called before the tainted data reaches the sink, **puts**. In this situation, one would expect a taint error to have been raised, and Quandary succeeds in doing so.

The same code, when run with the conditional's expression set to '1' rather than '0', produces no taint error, also as expected. Again, this matches our intuition, since the expression evaluating to true would result in the enclosed code being included for compilation by the pre-processor, which means the sanitizer would get called on the tainted data as required and prevent the occurrence of a taint error.

C. SOME EXAMPLES OF QUANDARY'S LIMITATIONS

1) Quandary fails to detect taint error when not all of the tainted data in a buffer is sanitized:

```c
#include <stdio.h>
#include <stdlib.h>

char *check(char *b) {
    return b;
}

void f() {
    char *buf = malloc(100);  // allocate 100 byte buffer
    if(buf == 0) return;

    gets(buf);

    buf += 99;  // point buf to last byte
    buf = check(buf);  // sanitize last byte
    buf -= 99;  // point buf back to first byte
    puts(buf);  // output tainted data to sink

    free(buf);
}
```

Listing 9. Quandary fails to detect a taint error when not all of the tainted data in a buffer is sanitized

Output:

```
No issues found
```

The pointer **buf** is incremented by 99 before calling the **check** function, which means that only the last byte is sanitized. After the call to **check**, the pointer is decremented by 99 so that it points back to the start of the buffer and the function calls for **puts** and **free** succeed.

To our surprise, Quandary does not raise a taint error with this code. It probably should, because only sanitizing the last byte of a source buffer before passing the (majority still-tainted) data to a sink does not give us much of a security guarantee.

One might argue that a developer is unlikely to ever write this kind of code in the first place – why would someone be messing around with pointer values to the extent that they're accidentally sanitizing only parts of the tainted data? Well, the author would suggest that in monumental, complex modern code-bases all sorts of bugs and errors are possible, and given the existence of actors who will actively and persistently *seek* to exploit code in any way that is viable, we can not make any assumptions about the code that is under analysis.

2) Quandary fails to detect a taint error when tainted data is copied back into buffer after sanitizing:

```
#include <stdio.h>
#include <stdlib.h>

char *check(char *b) {
    return b;
}

void f() {
    char *buf = malloc(100);
    if (buf == 0) return;

    char *copied_buf = malloc(100);
    if (copied_buf == 0) {
        free(buf);
        return;
    }

    gets(buf);
    strcpy(copied_buf, buf);  // store copy of tainted data
    buf = check(buf);  // sanitize the tainted data in buf
    strcpy(buf, copied_buf);  // cpy original tainted data back to buf
    puts(buf);  // output tainted data to sink

    free(buf);
    free(copied_buf);
}
```

Listing 10. Quandary fails to detect a taint error when tainted data is copied back into buffer after sanitizing

Output:

```
 No issues found
```

When analyzing the code above, Quandary does not behave as expected and fails to produce a taint error. Intuitively, we would expect that copies of tainted source data must also be sanitized before reaching any sinks — otherwise we are still potentially allowing malicious code to exploit a vulnerability in our system. However, in this test case the entire tainted buffer is copied to a part of the heap referenced by the pointer **copied_bf**, and the (potentially malicious) data successfully reaches the sink **puts**, without any complaint from Quandary.

3) Quandary fails to flag a taint error when calling sinks within sources:

```
#include <stdio.h>

char *check(char *b) {
    return b;
}

int *puts_call_sink_within_source(const char *str) {
        return puts(str);
}

char *gets_call_sink_within_source(char *str) {
        str="MALICIOUS_STRING";
        puts_call_sink_within_source(str);
        return str;
}

void f() {
    char *buf = malloc(100);
    if (buf == 0) return;

        gets_call_sink_within_source(buf);
        puts_call_sink_within_source(buf);
        free(buf);
}
```

Listing 11. Quandary fails to flag taint error when calling sinks within sources

Output:

```
No issues found
```

Quandary does not appear to identify a taint error if a sink procedure (in this case, **puts_call_sink_within_sourc** is called and passed tainted data from *within* a source procedure.

In this example, calling **puts_call_sink_within_source** with tainted data *after* the call to **gets_call_sink_within_s** results in a taint error as expected. However, the call to **puts_call_sink_within_source** from *within* **gets_call_sink_within_source** does not produce an error, despite the fact that a user or code within the procedure could inject and return a malicious string.

This is somewhat unexpected and not incredibly ideal, as a developer might want to craft their own custom functions for obtaining user input and define these as sources, in which case Quandary would fail to identify tainted code/input reaching a sink from within this function itself.

4) Quandary fails to flag a taint error if any single byte of a tainted buffer is copied and not sanitized:

```
#include <stdio.h>
#include <stdlib.h>

char *check(char *b) {
    return b;
}

void f() {
    char *buf = malloc(100);
    if (buf == 0) return;

    char *single_byte;

    gets(buf);
    strncpy(single_byte, &buf[50], 1);
    buf = check(buf);
        //single_byte = check(&single_byte);
        //note: allowing this check fixes the error as expected

    puts(buf);
    puts(single_byte);

    free(buf);
}
```

Listing 12. Quandary fails to flag a taint error if any single byte of a tainted buffer is copied and not sanitized

Output:

```
  No issues found
```

Quandary fails to produce a taint error if a single byte or an arbitrary sequence of bytes from a tainted source is copied and not sanitized before reaching a sink. This is somewhat unexpected and undesirable behaviour: it would not make sense to allow any part of the tainted input to make it to a sink—what if it contained an escape character that produces unintended behaviour in the target system, for example? Note that it appears to make no difference which particular byte or bytes are copied — we tried various combinations to no-effect. https://www.overleaf.com/project/5bdba98be169aa5e1c51656f

5) Quandary fails to identify taint errors involving copies of tainted source data:

```
#include <stdio.h>
#include <stdlib.h>
```

```
char *check(char *b) {
    return b;
}

void f() {
    char *buf = malloc(100);
    if (buf == 0) return;

    char *copied_buf = malloc(100);
    if (copied_buf == 0) {
        free(buf);
        return;
    }

    gets(buf);
    strcpy(copied_buf, buf);
    buf = check(buf);

    puts(buf);
    puts(copied_buf);

    free(buf);
    free(copied_buf);
}
```

Listing 13. Quandary fails to identify taint errors involving copies of tainted source data

Output:

```
No issues found
```

Interestingly, Quandary does not produce a taint error when analyzing the above code, but it probably should. Although the tainted data in **buf** is sanitized, we then overwrite it with a copy of the original unsanitized data which is stored in **copied_buf**, rendering the call to the sanitizer completely redundant.

We cannot make any assumptions as to whether a developer might accidentally (or perhaps even intentionally) copy tainted data around which ends up in the wrong place, so we see this as a rather substantial failure of the program.

6) Quandary fails to flag taint errors correctly in code with conditional statements: Quandary does not appear to be able to handle the analysis of source code nested within conditional statements. For example, if the call to a sanitizing function is nested within an **if** block which always evaluates to false, Quandary will fail to recognize that the code that calls the sanitizing function will never execute, which means that no taint error is produced, which would be the expected behaviour. Although the author is not aware of any existing literature on this specific topic, currently this deficiency is being investigated and formalised by Ernst & Murray.

VII. FUTURE RESEARCH

A. *Implement fixes for the limitations discovered in this research*

Of course, an obvious improvement upon this research would be to implement fixes for the limitations described and have them merged into Infer's main codebase.

B. *Create more test cases and quality documentation*

In addition to the above, researchers could conduct further investigation into Quandary's behaviours and create more test cases to illustrate these along with associated documentation and any potential or known solutions. Quandary's documentation is sparse at best, and it would be extremely useful to have quality documentation in place which describe its capabilities and limitations in detail. It would also be useful to document precisely how Quandary is implemented, in particular, how it leverages ideas from modern logic systems (e.g. separation logic, bi-abduction) to perform its analysis, and how these concepts and techniques are expressed in the source code.

C. *Extend Quandary using recent advances in information flow verification*

Another potential avenue for future research would be to extend Quandary using recent advances in information flow verification. Probably the biggest challenge would be to ensure that the information-flow control analysis maintains *compositionality*: one of Infer's main advantages is that rather than analysing the entire program at once, incremental changes to individual modules in the source code can be analysed and combined with a cache of results from the rest of the program in order to speed up analysis. However, as Hedin and Sabelfeld note, '...compositionality is typically not an intrinsic property of security definitions. Rather, compositional security properties that imply the weaker non-compositional security property are frequently formulated to enable proofs of correctness of compositional enforcement methods' [5].

D. Ensure that shared memory concurrent applications are analyzed correctly

A possible obstacle to the success of any future work would be in handling applications with shared-memory concurrency. Programs or parts of programs which are verifiably secure individually may fail to retain this property when run in parallel. Fortunately, recent research by Murray et al. [7] provides logics such as COVERN which can be used to formally verify information-flow control security of shared-memory concurrent programs. Specifically, COVERN can be used to prove the security property of timing sensitive noninterference. As discussed earlier in this report, timing sensitive noninterference is a stricter definition of noninterference than used traditionally, as it implies not only that secret data cannot affect public data, but also that secret data cannot impact the times at which public data is updated (otherwise an attacker could deduce the secret data by observing the cadence of the public data's updates). Also of significance is the fact that COVERN allows for compositional verification, i.e. each component of a program can be verified separately, with the end result that once all components are verified, so too is the entire system.

E. Perform the verification in C directly

Given that Quandary already supports the C language, we could also look at using recent work in formally verifying information-flow control for C directly. This is preferable to proving security properties over abstract specifications, since such proofs must still be transferred to real-world implementations, a process which can be relatively expensive [8].

VIII. CONCLUSION

Formal verification of software security is not only useful, but fast becoming *necessary* as code-bases become larger, more distributed, and more complex. But this powerful tool's usefulness depends on us first accurately modelling the systems we are trying to prove secure — a task which history shows is much easier said than done.

For a system to be secure, we need it to maintain the property of noninterference, i.e. that private variables do not affect the output of public variables. There are various types of noninterference, each of which can provide different guarantees as to the security of a system. For example, an important distinction exists between termination- vs progress-sensitive noninterference, which describe whether or not a program can leak information through to an attacker by observing its termination behaviour (i.e.

whether it has terminated or divered (i.e. is looping endlessly)), or by observing intermediate steps in the computation.

Static taint analysis approximates reasoning about noninterference and is a well-known technique which developers can take advantage of to help improve the quality, security and integrity of their code-bases. A static taint analyzer attempts to trace the flow of potentially malicious input from a developer-defined source through a system and ensure it is sanitized before being output to a sink. Static taint analyzers offer a reasonable trade-off between security and practicality.

Facebook's experimental module Quandary attempts to do static taint analysis, and throughout the course of this research we created a set of test cases which illustrate some of Quandary's capabilities and deficiencies in analyzing the C programming language. Specifically, we have discovered and documented the following[5]:

Capabilities

1) Quandary correctly recognizes when tainted data is sanitized before reaching a sink
2) Quandary correctly allows sanitizing functions to be called from within other functions
3) Quandary is able to correctly analyze code within C pre-processor conditionals

Limitations

1) Quandary fails to detect a taint error when not all of the tainted in a buffer is sanitized
2) Quandary fails to detect a taint error when tainted data is copied back into a buffer after sanitizing it
3) Quandary fails to flag a taint error when calling sink from within sources
4) Quandary fails to flag a taint error if any single byte of a tainted buffer is copied and not sanitized
5) Quandary fails to identify taint errors involving copies of tainted source data

In undertaking this research, we hope to have successfully documented some of the behaviour of a potentially useful tool for modern secure software engineering, and to have motivated further research into its effectiveness as a catalyst for engineering highly secure software. As a side-effect, we also hope that significantly more interest is generated in formal verification of secure software in general, as it is essential that we gain a deeper understanding of this field if we are to have any hope of surviving as a species into the Cyber Age.

[5]Note that the ratio of limitations to capabilities observed in the results of this research is no reflection on the overall quality of Quandary as a tool: if anything we were more interested in documenting its deficiencies and this influenced the way in which we approached testing the program, likely resulting in an over-representation of the tool's deficiencies in the data.

IX. REFERENCES

REFERENCES

[1] D. McCandless, P. Doughty-White and M. Quick, "Million Lines of Code," 24th September 2015. [Online]. Available: https://informationisbeautiful.net/visualizations/million-lines-of-code/.

[2] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick and D. Rodriguez, "Moving Fast with Software Verification," in NASA Formal Methods Symposium, 2015.

[3] C. Calcagno, D. Distefano and P. O'Hearn, "Open-sourcing Facebook Infer: Identify bugs before you ship," 11 June 2015. [Online]. Available: https://research.fb.com/open-sourcing-facebook-infer-identify-bugs-before-you-ship/.

[4] T. Murray and P. van Oorschot, "BP: Formal Proofs, the Fine Print and Side Effects," in IEEE SecDev 2018, 2018.

[5] D. Hedin and A. Sabelfeld, "A Perspective on Information-Flow Control," in Software Safety and Security, vol. 33, IOS Press, 2012, pp. 319-347.

[6] D. E. Denning, "A Lattice Model of Secure Information Flow," Communications of the ACM, vol. 19, no. 5, pp. 236-243, 1976.

[7] T. C. Murray, R. Sison and K. Engelhardt, "COVERN: A Logic for Compositional Verification of Information Flow," in EuroS&P, 2018.

[8] S. Gruetter and T. C. Murray, "Short Paper: Towards Information Flow Reasoning about Real-World C Code," in PLAS@CCS, 2017.

[9] Daniel Schoepe, Musard Balliu, Benjamin C. Pierce†, and Andrei Sabelfeld, "Explicit Secrecy: A Policy for Taint Tracking," in Conference: 2016 IEEE European Symposium on Security and Privacy (EuroS&P), March 2016,

[10] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez, "Moving Fast with Software Verification," Lecture Notes in Computer Science book series (LNCS, volume 9058), 08 April 2015,

[11] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee,Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), 14-18 June 2014,

[12] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," pp. 213-226, 2018 USENIX Annual Technical Conference (USENIX ATC 18), 2018.

[13] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom, "Spectre Attacks: Exploiting Speculative Execution", https://spectreattack.com/spectre.pdf (last accessed 2018-11-04),

[14] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, "Meltdown: Reading Kernel Memory from User Space," "27th USENIX Security Symposium (USENIX Security 18)", 2018,

[15] William G.J. Halfond and Alessandro Orso, "Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks," ACM SIGSOFT Software Engineering Notes 30(4):1-7, July 2005,

[16] Gerwin Klein, June Andronick, Matthew Fernandez, Ihor Kuz, Toby Murray, Gernot Heiser, "Formally Verified Software in the Real World", Communications of the ACM, October 2018, Vol. 61 No. 10, Pages 68-77,

[17] Scott Moore, Aslan Askarov, Stephen Chong, "Precise Enforcement of Progress-Sensitive Security," Proceedings of the 19th ACM Conference on Computer and Communications Security, October 2012,

[18] Vineeth Kashyap, Ben Wiedermann, Ben Hardekopf, "Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach," 2011 IEEE Symposium on Security and Privacy, 22-25 May 2011.

[19] Andrei Sabelfeld , Andrew C. Myers, "Language-Based Information-Flow Security," IEEE Journal on Selected Areas in Communications, 2003,

[20] Peter O'Hearn, John Reynolds, Hongseok Yang, "Local Reasoning about Programs that Alter Data Structures," International Workshop on Computer Science Logic, CSL 2001: Computer Science Logic pp 1-19,

[21] J.C. Reynolds, "Separation logic: a logic for shared mutable data structures," Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, 22-25 July 2002,

[22] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, Hongseok Yang, "Compositional Shape Analysis by Means of Bi-Abduction," Journal of the ACM (JACM) JACM Homepage archive, Volume 58 Issue 6, December 2011, Article No. 26.

[23] Tobias Nipkow, Lawrence C. Paulson, Markus Wenzel, "Isabelle/HOL: A Proof Assistant for Higher-Order Logic," Lecture Notes in Computer Science (Book 2283), 2002,

[24] Tobias Nipkow, "Teaching Semantics with a Proof Assistant: No more LSD Trip Proofs", Part of the Lecture Notes in Computer Science book series (LNCS, volume 7148),

[25] Adam Chlipala, "Modular development of certified program verifierrs with a proof assistant", ICFP '06 Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, pp. 160-171,

[26] Matteo AvalleAlfredo PirontiRiccardo Sisto, "Formal verification of security protocol implementations: a survey", Formal Aspects of Computing, January 2014, Volume 26, Issue 1, pp 99–123,

[27] Reynald Affeldt, Nicolas Marti, "Towards formal verification of TLS network packet processing written in C", PLPV '13 Proceedings of the 7th workshop on Programming languages meets program verification, pp. 35-46,

[28] Perl 5 Porters, "perlsec", perldoc.perl.org - Official documentation for the Perl programming language,

[29] The Open Web Application Security Project, "OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks," https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf (last accessed 2018-11-09).

X. APPENDICES

APPENDIX

A. Statement

I certify that this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.

Where necessary I have received clearance for this research from the University's Ethics Committee and have submitted all required data to the school.

The thesis is 6085 words in length (excluding text in images, tables, bibliographies and appendices).

Signed: [illegible] Date: 5/11/2018

B. Test Case GitHub Repository

https://github.com/srmcleman/infer-quandary-ifc