The Convergence of Source Code and Binary Vulnerability Discovery – A Case Study

Alessandro Mantovani
EURECOM
France

Luca Compagna
SAP Security Research
France

Yan Shoshitaishvili
Arizona State University
USA

Davide Balzarotti
EURECOM
France

ABSTRACT

Decompilers are tools designed to recover a high-level language representation (typically in C code) from program binaries. Over the past five years, decompilers have improved enormously, not only in terms of the readability of the produced pseudocode, but also in terms of similarity of the recovered representation to the original source code. Albeit decompilers are routinely used by reverse engineers in different disciplines (e.g., to support vulnerability discovery or malware analysis), they are not yet adopted to produce input for source-code static analysis tools. In particular, source code vulnerability discovery and binary vulnerability discovery remain today two very different areas of research, despite the fact that decompilers could potentially bridge this gap and enable source-code analysis on binary files.

In this paper, we conducted a number of experiments on real world vulnerabilities to evaluate the feasibility of this approach. In particular, our measurements are intended to show how the differences between original and decompiled code impact the accuracy of static analysis tools.

Remarkably, our results show that in 71% of the cases, the same vulnerabilities can be detected by running the static analyzers on the decompiled code, even though for several cases we observe a steep increment in the number of false positives. To understand the reasons behind these differences, we manually investigated all cases and we identified a number of root causes that affected the ability of static tools to 'understand' the generated code.

CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; *Vulnerability scanners.*

KEYWORDS

decompiler, SAST, vulnerability, reversing

ASIA CCS '22, May 30–June 3, 2022, Nagasaki, Japan.

ACM ISBN 978-1-4503-9140-5/22/05…$15.00
https://doi.org/10.1145/3488932.3497764

ACM Reference Format:
Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery – A Case Study. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30–June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3488932.3497764

1 INTRODUCTION

As our world continues to rapidly accelerate into a software-powered future, vulnerabilities in the software that increasingly supports our lives and livelihoods are on the rise. This poses a set of unique challenges for software development and testing. Software tends to be checked for bugs by two categories of testers: by those developing it and thus having access to the source code (source-level program analysis) and by external security researchers who, often, do not have access to the source code (binary-level program analysis).

Source-level vulnerability analysis is fundamentally different from binary-level vulnerability analysis, because critical information about the software, such as type, structure, and size information, is lost when the software is compiled. This makes performing certain analysis paradigms, such as static vulnerability detection, on binary code a daunting challenge: before vulnerabilities can be detected in binary code, this lost information must be somehow recovered. This explains why little work exists in this direction [7] and why commercial tools that can analyze binary code (such as Veracode) require the application to be compiled with debugging symbols [18] (i.e., inherently requiring the source code). Lack of source code also hampers other analysis paradigms, such as fuzzing and symbolic execution, because even these techniques benefit from the ability to *compile*, rather than retrofit, instrumentation into the analysis target [55]. As a result, static analysis techniques tend to require source code to effectively detect vulnerabilities, and dynamic techniques also function better when source code is available.

Interestingly, there is a related area of research that concerns itself with recovering information lost in the compilation process: *decompilation*. In recent years, techniques have been proposed to improve the recovery of data types [46, 53], code structure [35, 63, 64], and even exact syntactic identity [57]. These techniques have been integrated into increasingly powerful, accurate, and publicly available decompiler prototypes [33, 39, 40].

Our insight is that the place, conceptually, where decompilation leaves off is *close* to the place where vulnerability detection picks up. That is, we realized that the type information, structure

information, and pseudocode recovered by decompilers could be analyzed by vulnerability detection tools in lieu of the original source code, to at least some degree of efficacy. Additionally, as emerging techniques continue to improve decompilation results, and the gap between the original code and pseudocode from the decompilation of the program binary narrows, decompilers can become a more and more effective "crutch" to source-based vulnerability detection techniques.

In this paper, we undertake a study to determine the ability of current *Static Application Security Testing* (SAST) tools to detect vulnerabilities when executed on decompilers' generated code. While it might seem obvious that decompiled code is still unsuitable for static analysis, our case study wants to quantify experimentally how "far" we are from the point in which static analysis tools could be an effective solution on decompiled code. To do this, we measure precision and recall of 8 state-of-the-art SAST tools as they operate on the original code of 9 real-world applications versus the pseudocode of those applications resulting from the decompilation by 3 different state-of-the-art decompilers.

Our study has resulted in four main findings. First, the *output of current decompilers is unsuitable for any analysis by most SAST tools* without human analyst intervention and must be fixed before compilation-based analyzers (e.g. such as those based on LLVM passes) can be applied. Second, when the compilation issues are manually fixed, SAST tools operate at a reduced 71% rate of recall, suggesting that a latent potential could actually exist in the couple decompilers/SAST. Unfortunately, the precision of SAST tools on decompiled code suffered, with an average false positives increase of 232%. Third, we discovered that compiler optimizations, and especially function inlining, can sometimes *help* (and, at other times, hamper) SAST tools. Fourth, by analyzing discrepancies in SAST results between original and decompiled code, we identified and described 7 root causes that impact the differences between false positive and true positive detection performance.

In turn, we envision a number of immediate steps forward that can be inspired by our results. Our research solidifies an understanding that *modern decompilers are designed to generate code that is easy to understand for humans*, while SAST tools are not designed to ingest such machine-generated code. This suggests a set of new directions for researchers: even small improvements to decompilers can drastically improve the efficacy of SAST tools on *binary* code, despite the fact that they were designed with a source code requirement in mind. Alternatively, future studies could focus on SAST tools to make them more noise-resilient when parsing decompiled code. For example, the fuzzy-parsing approach performed by Joern [65], one of the SAST tools that performed well in our study, already goes in this direction. Furthermore, the use of decompilers as a first stage in source-level static analysis can have applications beyond the use of SAST tools on our dataset. For example, embedded device firmware remains difficult to test with either dynamic (due to the difficulty in executing the firmware without emulating specific hardware environments, known as the *rehosting problem*) and static (due to the binary-only form in which the firmware is often distributed) techniques. Though some limited progress on both fronts has been made [27, 41, 54], decompiler-aided static analyzers could automate vulnerability assessment for these scenarios where no standard alternatives exist.

In summary, this paper makes three main contributions:

- We "connect the dots" between decompilation and source-level static analysis, realizing that the latter picks up close to where the former leaves off.
- We perform a thorough evaluation of real-world applications with known vulnerabilities, measuring the resulting change in detection efficacy.
- We analyze the *root causes* of problems arising from SAST analysis of decompiled code, and propose concrete improvements that can be made by decompilation technique researchers to improve end-to-end SAST results.

All artifacts associated with the paper are present at the URL https://github.com/elManto/SAST_on_Decompilers.

2 RELATED WORK

We present now the state of the art related to both static application testing and decompilation.

2.1 SAST

Static Application Security Testing (SAST) aims at removing vulnerabilities from the source code during the development phase as argued by Chess et al. [25].

The first approaches proposed in this research field consisted of a simple lexical analysis of source code designed to detect the presence of known vulnerable constructs [16, 56, 60] (e.g., dangerous API invocations).

To overcome the limitations of these naive techniques, researchers proposed new approaches that leveraged a more detailed model of the application's source code, often obtained by relying on the compilers parsing components. For example the authors of [47, 59, 66] propose different methodologies to extract the AST of the source code at compile-time and use it for vulnerability detection.

Other researchers preferred instead to try to improve the detection accuracy for specific classes of bugs. This was the case, among the others, for buffer overflows [34, 36, 42, 61, 62], use after free [32, 67–69], and null pointer dereference [37, 38, 50].

That being said, our paper does not present a new static analysis approach. Rather, it is closer to the many studies that focus on benchmarking program analysis tools, such as [20, 21, 24, 30, 31, 43, 52], where several aspects are analyzed ranging from the creation of a comprehensive testcase to a different set of tools adopted in the experiments.

2.2 Decompilers

One of the first studies about decompilers was conducted in 1995 by Cifuentes [26], as part of her Ph.D. dissertation where she described how a decompiler works, the future challenges in the field and presented *dcc*, a decompiler for Intel 80286.

Over the past two decades, two main approaches have emerged for the development of decompilers: rule-based decompilation and NMT-based (Neural Machine Translation) decompilation. Rule-based approaches [10, 11, 23, 44] are the most popular today even though the production of a rule-based decompiler is particularly time consuming. For instance, according to its authors, the development of RetDec took a total of 7 years for a team of 24 developers [1].

The birth of the NMT-based approaches [33, 39, 40] coincides with the seminal work of Katz et al. [39], where the authors generalize the decompilation problem as a language translation task, namely from assembly to C thanks to the adoption of Natural Language Processing (NLP).

Another line of research has focused on improving the quality of the generated decompiled code by focusing on two main aspects: improving the readability and improving the control flow layout. The first category includes work that aimed at better recovering variable types [46, 53] and at suggesting more meaningful variable names [45]. The second category traditionally focused on reducing the number of GOTO statements generated by the decompiler [35, 63, 64] (*DREAM*/*DREAM++* decompilers) .

It is important to underline that all these studies focused only on improving the readability (and therefore the usability of the decompiler output) for humans. No study to date has analyzed how easy it is for a machine to process the produced code.

Finally, in 2018 Schulte et al. [57] proposed a novel approach to generate a binary-equivalent decompiled code that can be successfully recompiled. The paper by Schulte et al. relies on a number of innovative techniques, such as adoption of existing decompilers to seed the lifting process and use of a human-written code excerpts for the generation of human-readable code even though the tool (named BED) is not released.

Table 1: The vulnerabilities adopted for the evaluation

Vulnerability	Application	Description
CVE-2017-1000249	file (C)	Stack BOF, unchecked memcpy
CVE-2013-6462	Xorg compnent (C)	Stack BOF, unchecked scanf
BUG-2012	libssh2 (C)	IOF (leading to heap BOF)
CVE-2017-6298	ytnef (C)	NPD
CVE-2018-11360	wireshark (C/C++)	Heap BOF (off-by-one) (*)
CVE-2017-17760	OpenCV component (C++)	BOF in C++ virtual method
CVE-2019-19334	libyang (C)	Stack BOF, unchecked strcpy
CVE-2019-1010315	wavpack (C)	DBZ
BUG-2010	libslirp (C)	UAF
BUG-2018	wireshark (C/C++)	DF (*)

(*) indicates an inter-procedural bug, all the others are intra-procedural

3 METHODOLOGY AND EXPERIMENT DESIGN

This paper studies how modern static analysis tools are impacted by the decompilation process, from the perspective of vulnerability detection. To that end, we study the interaction of the following entities: SAST tools (Sec. 3.2), Vulnerable applications (Sec. 3.1) and Decompilers (Sec. 3.3).

For each vulnerable application, we proceed as summarized in Figure 1 (reported in Appendix for space reasons), where two main pipelines are executed.

Baseline analysis. In the *source code analysis* pipeline, we input the original source code of the application to the different static analyzers and store their generated reports for later analysis.

Compilation. We compile each application according to the provided build scripts (e.g., Makefiles), using the same compiler options as suggested by the developers, to obtain the *compiled binary* that is in turn fed into the *decompiled code analysis* pipeline. A further insight is presented in Appendix 4.8, where we show the results of the differential analysis we performed for a subset of the vulnerable applications to assess the impact of compiler optimizations.

Decompilation and analysis. In the decompiled code analysis pipeline, we decompile the binary using our decompilers and run the resulting code through the SAST tools that do not require recompilation.

As we will describe in more details in Section 3.2, the majority of the SAST tools require to compile the target application (for example, to perform LLVM passes). Therefore, since the decompilers typically generate C-like pseudocode which cannot be re-compiled out of the box, we manually applied the fixes needed to make the decompiler result compilable by both the `gcc` and `clang` compilers. This time-consuming process is interesting for different reasons. First, it allowed us to complete the experiments with all the static analysis tools selected in our study. Moreover, it provided us with an invaluable feedback on the steps an analyst should take if they want to apply source-code static analysis on binary programs. In other words, it allowed us to quantify the feasibility and effort required by a *human-in-the-loop* solution.

After manually repairing the decompiled results, we process the *recompilable* code by the compilation-based SAST tools.

Result comparison. Finally, we proceed to manually compare the three sets of reports obtained in our experiments (the one on the original source code, and the two on the decompiled and recompilable code) to assess how the detection and false positive rates were affected by the previous steps. The results of this comparison are presented in Section 4.

Whenever results differ (i.e., if a previously detected vulnerability was no longer detected or if new false alarms were generated by the tools), we performed a *root-cause analysis* to determine the cause. This step, again performed manually, required us to progressively modify the decompiled code by making it more and more similar to the original source, until the effect we wanted to study disappeared (i.e., the vulnerability was detected or the false alerts were not raised anymore). We discuss the findings of this analysis in Section 5.

In the rest of this section we discuss the methodology we used to select vulnerable applications, SAST tools, and decompilers. It is important to note that the applications and SAST tools had to be selected together. In fact, in order to have enough results for our comparison, we required each vulnerability to be detected by at least two SAST tools, and each SAST tool to detect at least two vulnerabilities.

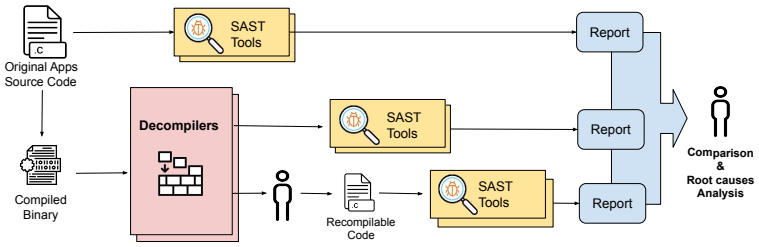

Figure 1: Summary of the Experiment Pipelines

3.1 Vulnerability and Application Selection

Our selection of vulnerable code was driven by five main requirements.

Codebase size. We wanted to include a mix of small and large code bases to assess the impact of code complexity on both the decompilation and the vulnerability detection phases.

C++. We included a C++ codebase to evaluate the fact that decompilers only produce C code as output.

Real vulnerabilities. We wanted a collection of real-world CVEs and bugs that can properly represent the typical classes of bug. This would allow us to be as general as possible in the actual evaluation phase, without focusing on artificially generated vulnerabilities.

Bug complexity. Third, an important factor that affects the precision of static analysis is whether the bug that needs to be detected is *inter-procedural* (i.e., its discovery involves to go through multiple functions) or *intra-procedural* (i.e., it is self-contained in a single procedure) In our dataset, we wanted to include examples of both categories, with a preference for *intra-procedural*. In fact, the purpose of our testbed is not only to benchmark SAST tools, but to cover bugs with different detection complexity.

Bug discoverability. Finally, we were also limited by the fact that the selected vulnerabilities should be identified on the original source code by the selected SAST tools, in order to perform a comparison when observing the verdict on the decompiled code.

To satisfy all our constraints, we collected 10 vulnerabilities from nine different applications (summarized in the Appendix in Table 1). The applications ranged from 4 Thousands to 2.1 Millions LOC (all LOC statistics are reported in Table 4). Note that for two projects, namely `Xorg` and `OpenCV`, the vulnerability was present in a sub component of the application that could be compiled as an independent module. Our dataset covers the following five classes of vulnerabilities:

Buffer Overflow (BOF) are probably the most widespread class of vulnerabilities and this is why we decided to include five variations of this class in our evaluation, e.g., three incorrect uses of a buffer handling API (respectively `scanf`, `memcpy` and `strcpy`), an example of heap-based off-by-one buffer overrun (inter-procedural) and finally a further stack-based BOF, present in a C++ code base and located in the implementation of an abstract method from a parent class.

Table 2: SAST tools selected for our study

	Commercial / Open Source	Compilation required	User-provided queries
CPPCheck	Open Source	No	No
Joern	Open Source	No	Yes
Comm_1	Commercial	Yes	No
Infer	Open Source	Yes	No
Clang	Open Source	Yes	No
Ikos	Open Source	Yes	No
Code-ql	Open Source	Yes	Yes
Comm_2	Commercial	No	No

Integer Overflow (IOF) bugs are a common cause of undefined behaviors in software. Our dataset includes one example of IOF that affects the size of a dynamic memory allocation, and that therefore can lead to an heap BOF.

Null Pointer Dereference (NPD) bugs exist when a NULL pointer is de-referenced. We include one example of NPD in our data set: in this example the pointer is returned by a `calloc` invocation and it is stored inside the field of a structure. The bug is due to the fact that the caller fails to check the pointer validity.

Double Free/Use After Free (DF/UAF) vulnerabilities. On the one hand, we would expect that such vulnerable flaws are easier from the decompilation point of view, because the decompiler can reconstruct the use of a `free` without any type system/size problems. On the other hand, SAST tools that detect DF/UAF need to internally keep track of freed pointers and check all the subsequent pointer accesses. As a further layer of complexity, one of the two bugs (the DF), is the second of the two inter-procedural vulnerabilities.

Division By Zero (DBZ) is not a memory corruption bug, but it affected several real world software in the past and can be used as denial of service vulnerability.

3.2 SAST Tools Selection

As we were unsure about the effects of the decompilation process on the analysis performed by SAST tools, we wanted to evaluate a range of products relying on a diverse set of features and techniques. We initially identified twelve tools (nine open source and three commercials) based on their popularity according to the studies proposed by [24, 31, 43, 51, 58] and including non-academic sources such as [2, 3, 19].

Over the 12 candidate SAST tools, we selected those that were able to satisfy the selection criteria of detecting at least two of the vulnerabilities in our dataset (cf. Table 3 detailed in Section 4).[1] Finally, our collection of static analyzers, listed in Table 2, includes: **CPPCheck** [6] 2.1, **Joern** [14, 65] 1.1.95, **Infer** [13] 0.17.0 , **Scan-build** [17] 11.0, **Ikos** [12] 3.0, **Code-ql** [4] 2.2.4, **Comm_1** and **Comm_2**, two popular commercial tools that we have to anonymize for legal reasons.

Before selecting these eight tools we conducted a set of preliminary experiments, in which we tested many other SAST tools. Among the others, we considered Comm_3 (another popular commercial tool), Frama-C [9], CPACheck [22] and Flawfinder [8]. However, we discarded them because after executing on a subset of bugs, they did not show a sufficient detection rate and accuracy of the analysis.

3.3 Decompiler Selection

We selected three cutting-edge decompilers for our evaluation: HexRays 7.1 [11] (the state of the art commercial decompiler from IDA Pro), Ghidra 9.2 [10] (the leader open source decompiler), and Retdec 4.0 [44] (the emerging challenger).

Two main reasons influenced our choice of these three tools. First, other emerging alternatives are quite far behind in terms of precision and quality of the generated code. Furthermore, prior work about decompilers [35, 57, 63, 64] only focused on these three decompilers when performing their evaluations.

Non-decompiling lifters. Some tools, such as MCSema [28], can lift binary code directly to LLVM IR, in lieu of decompilation [49]. At first glance, these might be a usable route for applying compilation-requiring SAST tools on binary code. However, these tools perform only a subset of the analysis which are executed by decompilers, and, in fact, can be considered as the "first stage" of a decompilation process. As a result, their output will contain insufficient information compared to the result of a decompiler, making the resulting code unsuitable for SAST analysis. For example, bytecode produced by lifters does not contain debug information whereas SAST tools that work on top of an llvm pass typically need compiler-generated symbols. Though it would be possible to develop more sophisticated SAST tools that bridge the gap between the output of static lifters and their expected input, this is exactly what decompilers already do from the other direction.

4 EXPERIMENTS

In this section we discuss our experiments with particular focus on how the decompilation process affects the detection and false positive rates as a whole and for each tool individually. We leave the investigation of the reasons behind these results to Section 5.

4.1 Source code analysis

Table 3 reports the detection results of the eight SAST tools against the different vulnerabilities in our dataset when analyzing the original source code of the applications.

It is interesting to notice that, with the notable exceptions of Joern, Clang and Code-ql, the other tools are quite complementary in the bug detection, uncovering 2-4 bugs each and missing only two bugs overall (CVE-2017-17760 and BUG-2018).

The high detection rate of Joern and Code-ql is due to the custom query rules written by us and inspired from the examples and guidelines described by the authors of the two projects [5, 15]. Although our scope was not to generate a query that is sufficiently generic to cover the many possible scenarios for a certain class of vulnerabilities, we tried to put ourselves in the position of an analyst who does not know the bug a priori and this explains why the user-defined rules still generate a number of false positives. [2].

Even though our effort was to produce generic rules, it is unavoidable to introduce some bias. However, we believe that this is the only way to include the two analyzers, that represent the current state-of-the-art w.r.t. source-code static analysis in our study. Making the queries more generic to catch a broader set of vulnerabilities for a specific class of bugs would also result in a biased result, by increasing the false positives. The opposite strategy (i.e., extremely dedicated queries that only capture the bug under testing) would not be representative of rules that can be used in the real world, as the analyst does not know the vulnerabilities beforehand.

The remaining six analyzers included in our analysis were launched with their own set of rules and thus they do not introduce any bias in the experiment. In particular, we decided not to create custom rules for other tools (such as Comm_2 or Comm_1) as they are already shipped with a full set of rules that were sufficient to detect some of the vulnerabilities in our dataset.

4.2 Decompilation

All three decompilers were able to successfully decompile the nine binaries in our dataset, with the exception of two executions of RetDec which failed on the largest projects (Wireshark and OpenCV) due to LLVM errors. It is very difficult to measure how accurate the generated pseudocode is, or even how close it is to the original source code. To accomplish this task, we draw inspiration from the authors of [35, 64], who adopted lines of code and number of GOTO statements as the core metrics to compare the different decompilers outcome in their work. As a coarse-grained indicator, Table 4 reports a comparison of the lines of code. The output of HexRays was the smallest in most experiments, and in total resulted in 20.8% more LOCs with respect to the original source files. Ghidra's code was not too far (+26.2% over the original), while RetDec was considerably more verbose (+79.8% in the eight binaries in which it ran successfully).

Previous papers on decompilation often counted the number of GOTO statements as a metric of the 'quality' of the produced code. While quality was often used as a synonym for readability, and it is unclear whether this would have any affect on static analysis tools, a lower number of GOTOs could also be considered a sign of a more advanced decompiler. We noticed that all tools generated code containing many GOTOs, ranging from a minimum of 84 (HexRays on `ytnef`) to a maximum of 36,002 (HexRays on `Wireshark`). In average, HexRays generated one GOTO every 60.3 LOCs (of the original source), Ghidra one every 60.7, and RetDec one every 11.2 LOCs.

[1] Note that the low detection rate of some tools may just be due to their underlying strategy in minimizing the false positive rate.

[2] queries are presented in a dedicated repository at https://github.com/elManto/StaticAnalysisQueries

Table 3: A breakdown of the bug detections for SAST tools (when applied to original program source code).

	CPPCheck	Joern	Comm_2	Clang	Ikos	Infer	Code-ql	Comm_1
CVE-2017-1000249	v	v	v	v	–	–	v	v
CVE-2013-6462	v	v	v	v	–	–	v	–
BUG-2012	–	v	–	v	v	v	v	v
CVE-2017-6298	–	v	v	–	v	v	v	–
CVE-2018-11360	–	–	–	–	v	–	v	–
CVE-2017-17760	–	v	–	v	–	–	–	–
CVE-2019-19334	v	v	–	v	v	–	v	v
CVE-2019-1010315	v	v	–	v	–	–	v	–
BUG-2010	–	v	–	v	–	–	v	v
BUG-2018	–	–	–	v	–	–	v	–

Table 4: Lines of Code produced by different decompilers

Application	Original	HexRays	Ghidra	RetDec
file	14,056	18,012	18,296	24,114
Xorg	20,331	32,131	31,132	62,819
libssh2	22,322	26,806	33,186	37,531
ytnef	4,025	3,529	5,736	4,427
wireshark	2,110,822	2,345,564	2,444,145	*NA*
openCV	507,508	826,104	871,848	*NA*
libyang	102,750	104,789	98,886	151,049
wavpack	6,084	8,671	11,645	23,084
libslirp	7,806	12,178	14,058	15,915

Finally, we compared the function declarations of the projects source code against the ones contained in the pseudocodes produced by the three decompilers in order to measure the difference in the number of input parameters. On average, HexRays misses 4 parameters, Ghidra 6, and RetDec 7 every 10 function declarations.

4.3 Re-Compilation

Three among our SAST tools can directly analyze source code files without any need to compile them: CPPCheck, Joern and Comm_2. The first two were able to analyze the output of the decompilers, without any further manual intervention. Comm_2 instead failed at reconstructing the AST for five instances of decompiled code.

Furthermore, the remaining five tools require the compilation of the target application to analyse it. However, as shown by the authors of [48], *none* of the output produced by the three decompilers was correct C code, and therefore none of them could be re-compiled out-of-the-box. This obliged us to look for a suitable solution to continue our experiments.

Therefore, to put ourselves in the position of an analyst, we attempted to manually fix the produced pseudocode to make it compliant with both GCC and Clang. We performed this operation on the output of all the three decompilers considered in our study, to compare different executions of the static analyzers on different input pseudocodes.

Overall the manual procedure took from a minimum of 90 minutes to 8 hours (for `libyang`). However, after spending 24 hours each by trying to fix the decompiled code of Wireshark and OpenCV (the two largest projects), we could not obtain a "recompilable" version of the pseudocode. Hence, for these two applications we adopted an alternative solution, that allowed us to generate a version of the decompiled applications that preserved the vulnerabilities and could be processed by our SAST tools. In particular, for these two cases we fixed the pseudocode of the vulnerable functions and of all of the procedures they invoked. We then integrated the resulting code into the the original source code of the vulnerable module – thus resulting into an hybrid codebase where all code related to the vulnerability came from the decompiler while the rest was taken verbatim from the original codebase of the module. This compromise allows us to study whether SAST tools could still find the vulnerability in the recompilable code, extending our evaluation of the tools to all the pre-selected vulnerabilities, but not to measure the impact on the overall number of false positives.

Our manual procedure consisted of a number of repeated steps that involved the proper definition of global variables, the definition of header files, the correction of function invocations (e.g., often the decompiler declared a method with N parameters and invoked it with M ! = N parameters), the resolution of mismatching types, and some small syntactic operations to remove wrong keywords or fix syntax errors with brackets.

Although we are aware of the fact that some bias could be introduced while manually fixing the pseudocode, we want to underline that this mimics a realistic setting since currently a human-in-the-loop solution is required for this approach and alternatives are still missing.

4.4 Decompilers variability

The detection outcomes of the SAST tools able to analyze the output of the three decompilers is presented in the 'Decompilers Output' columns of Table 6. These outcomes are not broken down for each of the three decompilers as, except for the case of CVE-2017-6298 discussed below, the detection results were always the same regardless of the decompiler.

Indeed we launched the 8 static analyzers for each version of the decompiled code (either the raw or the manually fixed one depending on the tool). Unfortunately, some combinations of analyzer-pseudocode could not produce an analysis result because the corresponding tool failed with a crash. Except for an execution of Ikos on the Hex-Rays decompilation of CVE-2019-1010315, the other exceptions affected mostly the output of Ghidra and Retdec when analyzed by Ikos (3 failures on Retdec, 5 on Ghidra), Comm_1 (2

failures on Retdec) and Comm_2 (3 failures on Retdec, 2 on Ghidra). For all the other tools instead, it was possible to compare the output in terms of detections, finding that no differencies exist between the HexRays and Ghidra outcome from the SAST perspective.

The same does not hold for the output of RetDec. Overall the code generated by RetDec was more complex and considerably less readable for a human analyst. However, readability does not necessarily affect automated algorithms, and in fact vulnerability CVE-2017-6298 could only be detected on the RetDec output when using Joern and Code-ql. This is due to the fact that RetDec adopts a more naive approach and represented the fields of a struct as if they were separate variables (while both Ghidra and HexRays reconstructed a struct), before assigning them in the pseudocode representation of the struct (i.e., an array). As we will explain in more details in Section 5, this helps static analysis tools to more easily track the use of the individual fields, which in the aforementioned case helped to discover the vulnerability.

We searched for other cases containing structs to see if they also benefited from the RetDec decompilation approach, but neither the Use-after-free nor the Double free bugs that are related to struct usage could be discovered on the RetDec decompiled code. Note that since RetDec failed to decompile Wireshark in its entirety, we manually tried to point the tools directly to the vulnerable functions (which were decompiled by RetDec), but this did not lead to any detection because in those cases the generated code was more similar to the HexRays one and it contains some patterns that make the bug detection harder. As we will explain more in details in 5, the representation of types and structs in the pseudocode is crucial for SAST tools.

In the rest of the paper we consider a bug as detected by a static analyzer on a binary if at least one decompiled code exists such that the tool can identify the vulnerable flaw when analysing it. Similarly, due to space limitations, for Table 6 (where we evaluate the variation of false positives), we only report results on the HexRays decompiled code. Moreover, given the failure conditions that some tools experienced on Retdec and Ghidra, the false positives evaluation on these would be incomplete.

4.5 Summary of Results: True Positives

Table 5 presents a summary of the results, both for the tools that we were able to run on the vanilla output of the decompilers, as well as for the remaining ones that we had to test on the manually curated code. The green marks represent the cases where the bug was found on the pseudocode, whereas the cross marks show a missing detection. Dashes instead indicate that the bug was not found in both the original source code and the decompiled one.

We must underline that for five executions on the raw HexRays decompiled code, Comm_2 failed at building the AST of the analyzed code. For this reason, we opted to run it on the re-compilable code and to report the results related to such executions.

Overall, only one of the tools (Chechmarx) was able to re-discover the same subset of vulnerabilities as when it was applied to the original source code. However, all tools were still able to discover at least one bug (and often more than one), thus showing that running SAST tools on decompiled code is not a useless procedure. In total, the 42 cumulative True Positives on the original codebase decreased to 30 (71%) after decompilation. However, not all tools were equally affected, as reported in the last row of the Table.

The three tools that operate on source code without the need to compile it were less affected by the decompilation process. Moreover, the commercial tools, while in general less effective at discovering the vulnerabilities in our dataset, continued to find exactly the same bugs also in the decompiled code, even though in the case of Comm_1, we can observe that a new vulnerability is uncovered instead of another that is not detected anymore. At the other end of the spectrum, Clang and Code-ql were the two tools that were affected the most by the decompilation process.

Another way to look at the data is to group the results in terms of vulnerabilities instead of looking at the different tools. In this case (all results reported in the last column of Table 5) the integer overflow (BUG-2012), the use-after-free (BUG-2010), and the double-free (BUG-2018) clearly stand out as the most difficult to detect on decompiled code.

At the other end of the spectrum, the division by zero and the stack-based buffer overflows seemed instead the easiest to detect. For the first, a manual inspection shows that there are no interesting variations in the way the decompilers reconstructed the source code. The bug involved two integer variables which are easier to handle than strings/pointers for decompilers. Thus, after decompiling the corresponding binary, the pseudocode surrounding the vulnerability was quite similar to the original code, from a static analysis perspective.

For the three stack-based BOFs, the true positive instead came at the expense of a much larger number of false positives, as we will describe in more detail in the following section. For these cases we reported an asterisk (*) meaning that an high number of buffers' operations were flagged by the analyzer, partially explaining the detection for these cases.

4.6 Summary of Results: False Positives

The usability of a tool is largely determined by the number of false positives, since reporting thousands of alarms would make the triaging phase both difficult and time consuming.

We performed a study of the false positive increment for each project where we could compare the outcomes of the tools on the decompiled code. Thus, we decided to focus on the Hex-Rays output, since is the one that was easier to parse for the SAST tools reporting only one failure for CVE-2019-1010315 (as explained in Section 4.4, 3 tools failed on the Ghidra/Retdec output). Moreover, it was not possible to have such a comparison on the Wireshark and OpenCV projects, because we could not recompile the decompiled code.

We report the variation of false alarms in Table 6, marking in red the cases in which there was an increase of more than 50% of false alarms, and in green when the number decreased. Overall, if we exclude Joern (which is a special case we describe below) in 78% of the tests the number of false positives increased. Even worse, in 61% of the tests the false alerts increased by more than 50%.

We point out that we manually went through the generated alarms that the static analyzers produced, to assess if they represented actual false positives. The only assumption that we did to accelerate the procedure, was that if the use of an API call (e.g.,

Table 5: Results of running the SAST tools over decompiled code. An asterisk (*) signifies that the detection was accomplished through the introduction of an excessive amount of false positives.

	Decompilers Output		Re-Compilable Code						Wrt the original
	CPPCheck	Joern	Comm_2	Clang	Ikos	Infer	Code-ql	Comm_1	codebase
CVE-2017-1000249	x	v	v*	v*	–	–	v	x	4/6
CVE-2013-6462	v*	v	v*	v*	–	–	v*	–	5/5
BUG-2012	–	x	–	x	v*	x	x	x	1/6
CVE-2017-6298	–	v	v*	–	x	v	v	–	4/5
CVE-2018-11360	–	–	–	–	v	–	x	–	1/2
CVE-2017-17760	–	v	–	v	–	–	–	–	2/2
CVE-2019-19334	v*	v	–	v*	v*	–	v	v*	6/6
CVE-2019-1010315	v	v	–	v	–	–	v	–	4/4
BUG-2010	–	x	–	x	–	–	x	v	1/4
BUG-2018	–	v	–	x	–	–	x	v	0/2 + 2
Wrt original codebase	3/4	6/8 + 1	3/3	5/8	3/4	1/2	5/9	2/4 + 1	

Table 6: False positives, as evaluated on the original source code and the decompiled source code produced by Hex-Rays. False positive increases of over 50% are highlighted in red, and decreases are highlighted in green.

	Decompilers Output				Re-Compilable Code											
	CPPCheck		Joern		Comm_2		Clang		Ikos		Infer		Code-ql		Comm_1	
	Src	Dec	Src	Dec	Src	Dec	Src	Dec	Src	Dec	Src	Dec	Src	Dec	Src	Dec
CVE-2017-1000249	68	45	166	144	236	241	88	96	3	9	152	423	174	515	202	583
CVE-2013-6462	106	503	374	296	485	701	368	365	303	359	89	812	496	1,177	246	1,097
BUG-2012	68	262	253	110	107	358	298	178	103	541	21	827	102	341	613	453
CVE-2017-6298	16	125	25	39	52	66	15	10	329	439	10	80	18	132	84	137
CVE-2018-11360	3,113	20,548	3570	1504	-	-	-	-	-	-	-	-	-	-	-	-
CVE-2017-17760	475	7668	122	1277	-	-	-	-	-	-	-	-	-	-	-	-
CVE-2019-19334	833	1072	1019	334	2,295	384	185	315	13	113	128	959	98	176	1999	661
CVE-2019-1010315	20	143	214	150	173	358	65	45	20	-	295	361	428	208	1417	806
BUG-2010	69	297	190	41	102	114	2	22	31	32	23	168	37	350	115	525
BUG-2018	3,113	20,548	3570	1504	-	-	-	-	-	-	-	-	-	-	-	-
Total	7,881	51,211	9,503	5,389	3460	2223	1,021	1,031	802	1439	718	3630	1353	2899	4676	4262

`strcpy` or `memcpy`) was safe in the source code, it could not become vulnerable in the pseudocode. Moreover, many false positives could be discarded in batch since they were related to uninitialized variables.

However, in some cases (mainly for Clang and Comm_1) the tools generated less false alarms on the decompiled code. To figure out the reasons behind this, we checked the reports for those tools that reported a negative variation. One of the main reasons for this behavior is that many false alarms in the source code are due to `free`-related vulnerabilities (UAF, DF, stack variables freed). However, when analysing the decompilers, the SAST tools could not apply the same dataflow and, moreover, the decompiler changed the type of the variable containing the freed memory area, making the job of the analyzers much more difficult. Furthermore, several warnings reported the presence of badly terminated strings in the source code (i.e., strings without the proper null-terminated byte). Because of type confusion problems, the same problem could not be detected in the decompiled code.

To evaluate the Code-ql false positive rate, we adopted the default queries that are shipped with the installation. This allowed us to obtain unbiased results, compared to what we would get if we used the custom rules that we wrote to find the vulnerabilities.

Finally, Joern deserves a separate discussion as the tool does not come with any predefined rule, and all tests were therefore performed by enabling our own heuristic checkers for each project scan. Although these are certainly not a complete and generic set, they allowed us to have a reasonable evaluation of the false positives also for this static analyzer. Moreover, such a tool performs a fuzzy parsing of the code. Even if this feature makes Joern the perfect candidate to analyze decompiled code, we pay for this fact in some cases where it could not correctly interpret some pieces of code and skipped them without providing a complete analysis. As a result, the internal representation lacks some parts that could not be correctly parsed, and thus were not reachable by our queries. This resulted in a decrease of the query output, as only a portion of the code could be properly analyzed.

4.7 Bugs detected *only* on pseudocode

Our initial assumption was that running a SAST tool on decompiled code can *at best* detect the same bugs it would detect when it is used to analyze the original source code of the application (and more likely considerably less than that). Albeit our experiments show that for the majority of the analyzed scenarios this hypothesis is correct, we found one interesting case (BUG-2018) in which tools (both Joern and Comm_1) could detect a vulnerability on the decompiled code, but not on the original codebase.

Compilers can affect the control flow of a program in such a way that it is impossible to recover exactly the original version. For example, as we will see in the next sub-section, sometimes compilers remove dead code or simplify boolean conditions for optimization reasons.

The double free vulnerability present in Wireshark (BUG-2018) is an inter-procedural problem, which is therefore harder to detect for static analysis tools. In fact, as reported in Listing 1 (that we report in the Appendix for space reasons), the vulnerability involves three separate functions that eventually invoke the `g_free` two times.

In the original codebase Joern was only able to reconstruct a subset of the flows that lead to `free`, thus missing the vulnerability. Similarly, the internal analysis performed by Comm_1 was insufficient to uncover the vulnerable flow in the original source code.

However, after checking the decompiled code, we noticed that, because of the `static` keyword, the compiler inlined different functions into a single body (`val_from_unparsed`). This transformed the inter-procedural bug into an intra-procedural one, largely simplifying the task of detecting the bug. In fact, it turned out that both Joern and Comm_1 succeeded at revealing the bug on the pseudocodes they could analyse.

```
1  static void
2  string_fvalue_free(fvalue_t *fv)
3  {
4      g_free(fv->value.string);
5  }
6
7  static gboolean
8  val_from_string(fvalue_t *fv)
9  {
10     string_fvalue_free(fv);
11     return True;
12 }
13
14 gboolean
15 val_from_unparsed(fvalue_t *fv, ...)
16 {
17         string_fvalue_free(fv);
18         ...
19         return val_from_string(fv, ...);
20 }
```

Listing 1: Double free source code

4.8 Compiler Impact

To verify the compilers' impact, in terms of optimization levels , we performed an additional experiment based on the following four phases. **(i)** Selection: we selected two among our open source projects, `file` and `libssh2` (CVE-2017-1000249 and BUG-2012). The choice of the two projects was driven by the average size of their codebase and the meaningful number of detections. **(ii)** Compile with optimization levels: we compiled the selected projects with three different optimization levels, `O0`, `O2`, `O4` (`O0` disables all optimization passes whereas `O4` indicates that the generated code is highly optimized to improve execution speed). It shall be noticed that all the experiments discussed so far were performed using the default compiler optimizations specified in each project makefile (always `O2` for our applications). **(iii)** Decompile: we decompiled the three versions of the same binary with HexRays. **(iv)** Analysis: we launched all SAST tools on each decompiled outcome. This also implies that we had to manually fix all variants of the decompiled code to generate the recompilable versions required by many of our tools.

The first aspect we wanted to investigate is how the compiler options affect the number of false positives. All static analyzers ran on all versions except for Ikos that reported some problems when parsing the code compiled with the `O4` option. Hence, we discarded it, for the computation of the false positives.

For `libssh2`, the tools cumulatively produced 850, 2,421 and 1,606 false positives for, respectively, `O0`, `O2` and `O4`. For `file` we obtained instead 3,085, 2,275 and 2,984 alarms, depending on the compiler optimization. Such results show that there is no clear trend and it is unclear whether a more aggressive optimization of the code would cause more or less false alarms. However, the different amount of false positives for each compilation option means that the compiler actually has an impact on the generated decompiled code, and therefore, on the way SAST tools parse it.

We then inspected all reports generated by the tools, to determine if vulnerability detection is also affected by the compiler optimizations. For BUG-2012, we could not find any difference between the executions of the static analysis tools over the different versions of decompiled code. The only configuration that brought to a detection consisted of executing Ikos on the `O0` and `O2` versions of the code. After a manual inspection of the three flavours of pseudocode, we understood that the compiler optimization level does not affect the vulnerable function in a significant way except for a different number of declared variables(29 for `O0` vs 99 for `O4`).

CVE-2017-1000249 instead tells a different story. Indeed, when scanning the three versions, the tools reported different results depending on the compiler optimization. More specifically, with `O0` and `O2`, 4 tools out of 8 could detect the bug anyway. Surprisingly, the detection dropped to zero with the `O4` flag. To understand the reasons behind such a drastic change, we went through the decompiled code one more time. A first difference is that with the `O4` flag, multiple functions are compiled inline and thus, the vulnerable function becomes a part of a bigger one, hindering the SAST tools to analyze the data flow. Moreover, such a modification, affects not only the local defined functions of the binary, but also some library functions. Among the others, the `memcpy` invocation, originally contained in the code and root cause of the buffer overflow is replaced

Table 7: Decompilation inadequacies that inhibited SAST tool operation on our dataset

Pattern	Effect	Project	Affected Tool	Repairer
(P1) Buffer size	FP ↑	all	all except Joern	Both
(P2) Integer types	FP ↑	all	Comm_2, Cppcheck	SAST
(P3) Unitialized variable	FP ↑	all	Clang, Infer, Ikos,Code-ql	Decomp
(P4) Function pointers	TP ↓	`libssh2`	Joern,Code-ql	SAST
(P5) Pointer as int	TP ↓	`ytnef`, `wireshark`, `libslirp`	Joern,Clang Comm_2, Ikos,Code-ql	Both
(P6) Int wrong size	FP ↑	all	Ikos,Code-ql, Infer	Both
(P7) Simplified expressions	TP ↓	`file`	Cppcheck Comm_1	SAST

with an inline implementation that is ignored by the tools. Finally, an unsafe check on the buffer size that always evaluates true (because of a programming error) is removed due to the optimization reasons, as described in more details in Section 5. Cumulatively, these three aspects make the life of SAST tools remarkably harder, leading to an increment of the false negative.

Although this experiment cannot uncover in a systematic way all possible scenarios where the compiler influences the resulting pseudocode, these observations indicate that the compiler influences the decompilation phase w.r.t. both the false positives and false negatives.

5 ROOT CAUSE ANALYSIS

In this section we conduct an investigation to figure out the reasons behind the differences between each SAST tool execution on source code and decompiled output. For this purpose we gradually change the pseudocode by making it more and more similar to the original codebase, until either the tool reported the missing vulnerability or until the extra false positive disappeared.

Our findings uncovered seven main root causes, four responsible for false positives and three for false negatives. For each of them we discuss the specific elements in the code (hereinafter *patterns*) introduced by the compilation and decompilation process that degraded the SAST performances. The list of patterns is summarized in Table 7, together with the projects and tools affected by that specific pattern. For each one, we indicate a **Repairer**, i.e., the component of the toolchain (decompiler, SAST tool, or both) that is in the best position to mitigate/address the problematical pattern. In fact, while on the one hand decompilers could try to infer more information from the binary, on the other hand SAST tools can be designed with this limitation in mind and be more permissive when dealing with the pseudocode.

Finally, we want to stress that our purpose is to illustrate such root causes that result in the performance degradation of the SAST tools, rather than proposing potential remediations to these issues. Indeed, as explained in the section, the reported problems are not trivial to address, and may require future research in both the decompilation and the static analysis fields.

P1 - Inability to Recover the Size of Stack Buffers
Effect: increase false alarms Repairer: Both

A large number of extra warnings in the SAST output was reporting the presence of buffer overflows. As an example, we propose the following excerpt from the `file` application:

```
#define PATH_MAX 4200
FILE* list = fopen(outfilename+1 "rb");
char listbuff[PATH_MAX * 2];
memset(listbuff, 0, sizeof (listbuff));
fread(listbuff, 1, sizeof(listbuff)-1, list);
```

Listing 2: Source code of a safe buffer access

Looking at this code, it is quite evident that the two memory write operations (i.e., `memset` and `fread`) are safe in this context, thanks to the proper use of `sizeof` operator. The decompiled code looks instead quite different:

```
FILE* v212;
char* s1;
v212 = fopen(dest + 1, "rb");
memset(s1, 0, 0x2000uLL);
fread(s1, 1uLL, 0x1FFFuLL, v212);
```

Listing 3: Pseudocode of a (ex safe) buffer access

The `sizeof` operator is resolved at compile-time, and therefore the decompiler only sees the actual numerical values. Intuitively, one would expect that this makes the SAST's job easier because now the tools do not need to compute themselves the size value. However, the array definition has been replaced with a scalar variable (`s1`) declared as a `char*`, without any information about its original size.

As a result, when the SAST tools analyze the decompiled code, they flag the two calls as two potential buffer overflows, since the memory area pointed by the `char* s1` variable has unknown size.

In other examples, different ways to access the buffer (e.g., by index `buf[i]`), resulted in different warning such as null pointer dereferences, still because of the missing size information of a pointer variable.

Discussion: Although the issue is quite evident when comparing source and decompiled code, a proper solution is not as simple and it inherently depends on the way compilers work and generate the assembly code. In fact, even with a sophisticated analysis of the stack, the decompiler cannot infer whether a memory area belongs to the same buffer or it represents a group of distinct variables (in particular when an element of a buffer is accessed by using an hardcoded index).

While some heuristics could be used to infer the original size, e.g., by looking at loop iterations or initialization routines, the risk is that by relying on this information the decompiler can hide the presence of vulnerabilities.

P2 - Signed and Unsigned Integers
Effect: increase false alarms *Repairer: SAST*

Another source of false positives was related to SAST tools flagging several numerical statements as potential IOFs. At a closer analysis, this was caused by two main errors in the decompiled code.

An example of this pattern are functions that return an integer value, where a negative value is associated with an error condition. For instance, this is a snippet of decompiled code from the `Xorg` project:

```
sub_9840(__int64 a1) {
  unsigned int v2;
  ...
  if(ERROR_CONDITION) { v2 = -1;}
  return v2; }
```

Listing 4: Negative return value in the pseudocode

The `v2` variable is used to store the return value and it is assigned to `-1` in case of an error condition. However, `v2` is erroneously declared by the decompiler as `unsigned int`, and thus, assigning a negative value, leads the SAST checkers to think that an underflow can occur within that variable.

P3 - Integer Operations on Uninitialized Variables
Effect: increase false alarms *Repairer: Decompiler*

Mainly due to a more complex and interprocedural data flow in the decompiled code, we noticed that many SAST tools reported an addition (or subtraction) as potentially dangerous when they could not determine if one of the operands has been initialized.

As an example, we fetched the following lines of code from the `libyang` pseudocode:

```
1  sub_12B3E (..) {
2    v2 = 10; v4 = 1;
3    .. // a lot of code including GOTOs, etc
4    sub_129CF(.., &v2, .., v4);
5  }
6  sub_129CF(.., unsigned __int16 *a2, .., unsigned
       __int16 a4)
7  {
8    unsigned __int16 v9 = a4;
9    ..
10   *a2 -= v9;
11 }
```

Listing 5: Integer underflow due to an uninitialized variable

The subtraction at line 10 is indicated as dangerous by Infer and Ikos , because both the tools cannot find an initialization statement for the operands. However, when we checked the source code, and compared against the decompiled code, we noticed that in both the two variables are initialized. The key difference is that in the source code those variables are initialized just before calling the function, while in the pseudocode they are initialized at the very beginning of the program so that very likely the SAST tools lose track of their propagation because of complex data flow.

Interestingly, so far decompilation researchers mainly studied variable types recovery [29, 46, 53] and names generation [45], but no prior work focused on the "position" of recovered variables in the control flow.

P4 - Function Pointers
Effect: decrease detection rate *Repairer: SAST*

BUG-2012, which affects `libssh2`, is presented by Yamaguchi et al. [65] as the typical use case for the adoption of Joern, but while such a tool can detect it on the original codebase, it misses its presence in the decompilers output. The main vulnerability consists of an integer overflow resulting from a sum whose value is used as input for a dynamic memory allocation. As a consequence, the IOF can produce an undefined dynamic memory allocation resulting in wrong memory accesses. For clarity, we report the snippet of code in the following listing:

```
ssh2_packet_add(SESSION session,char* data,..){
...
uint32_t namelen =
  libssh2_ntohu32(data+9+sizeof("exit-signal"));
channelp->exit_signal =
  LIBSSH2_ALLOC(session, namelen + 1);

memcpy(channelp->exit_signal,
       data+13+sizeof("exit-signal"), namelen);
}
```

Listing 6: libssh2 vulnerable code snippet

The `LIBSSH2_ALLOC` macro allocates `namelen + 1` bytes and returns the requested memory in the `exit_signal` buffer, that is eventually accessed. If `data` is under the attacker control, it is possible to craft that variable so that the sum `namelen + 1` causes an IOF bug.

Listing 7 shows the resulting pseudocode:

```
vulnerable_function(int64 a1, int64 a2, ...) {
...
int name_len = non_vuln_function(a2 + 21);
*(_QWORD *)(v24 + 40) =
  (*(__int64 (__fastcall **)(_QWORD, __int64))
  (a1 + 8))((int)(name_len + 1), a1);

memcpy(*(void **)(v24 + 40),
  (const void*)(a2 + 25), name_len);
}
```

Listing 7: libssh2 vulnerable decompiled code

Now we can notice that the macro invocation has been replaced with its actual value that corresponds to a function pointer stored in the `struct session` at offset 8 (namely, the macro is defined as `session->alloc(...)`). The decompiler casts the function pointer accordingly to the function definition, resulting in a more complicated structure of the invocation.

The pointer cast is the culprit of the problem and the reason Joern and Code-ql are ineffective against this code. The first of the two tools is not able to properly parse it, and thus it entirely skips the call. In this case no query exist that can reach the vulnerable path.

Code-ql actually parses the code correctly, but because of the internal representation used by the framework, the query used to

find the original vulnerability does not work anymore. It is possible to write a new, and much more generic, query that still capture the bug — but the more general rule would cause an increased number of false positives.

Discussion: The root cause of the problem is that the function pointer invocation contains many casting operations, therefore hindering the static parsing of the code. However, we could easily solve the problem by instantiating a variable that can store the function pointer address, and then invoking it in a separate line:

```
__int64 (*fcn_ptr)(_QWORD, __int64) = (a1 + 8);
*(_QWORD *)(v24 + 40) = (*fcn_ptr)((unsigned int)(
    n + 1), a1)
```

Listing 8: libssh2 vulnerable decompiled code after the fix

P5 - Pointers as Integers

Effect: decrease detection rate *Repairer: Both*

For this pattern, let us focus on the CVE-2017-6298, a null pointer deref resulting from an unchecked `calloc` return value.

Reading the following snippet of code the vulnerability looks quite evident, and in facts different tools can detect it (Joern, Comm_2, Ikos, Infer, and Code-ql):

```
variableLength* vl;
...
vl->data = calloc(vl->size, sizeof(WORD));
temp_word=SwapWord((BYTE*)d, sizeof(WORD));
memcpy(vl->data, &temp_word, vl->size);
```

Listing 9: Null pointer dereference

The first thing is that the variable `vl` is a pointer to a custom struct whose definition is unknown for the decompiler. The `memcpy` invocation itself is safe because the code writes the correct size into the dynamically allocated buffer, but the `vl->data` value is not checked for nullness, potentially leading to a null pointer dereference if `calloc` returns a null value.

When the code is decompiled with HexRays and Ghidra, we obtain the following code:

```
signed int* v9;
size_t v19;
void* v20;
...
(_QWORD *)v9[0] = calloc(v9[2], 2uLL);
v18 = sub_19B0(v4, 2);
v19 = v9[2]; v20 = *(void **)v9; v76 = v18;
memcpy(v20, &v76, v19);
```

Listing 10: Null pointer dereference Hexrays pseudocode

We can immediately note that the struct is represented as a signed integer pointer (identifier `v9`). The `calloc` return value is written in `v9[0]`, after casting it to pointer.

Although the tools comprehend that the return value is written inside a local variable, they believe that the assignment happens in a variable of type `signed int`. Because of such a type confusion problem, from now on the static analyzers are not interested anymore in the return value, stop tracking the data flow for that path and go on with the analysis of other potentially vulnerable paths. Overall, they miss the connection between the returned pointer and its dereferences that occur in the following code.

If we look instead at the code generated by RetDec:

```
int64_t * v103;
int32_t v105;
..
int64_t * mem5 = calloc(v102, 2);
*v103 = (int64_t) mem5;
int64_t v104 = fun_19b0(v19, 2, v28, v1);
int64_t v108 = v104;
...
memcpy((int64_t *)* v103, &v108, v105);
```

Listing 11: Null pointer dereference RetDec pseudocode

What makes this output simpler to analyze for static analysis tools is the fact that the return value of the `calloc` API is directly stored into a proper pointer, without any further cast or array access. Thus, tools are able to track the data flow and can therefore recognize the use of the pointer within the `memcpy`.

In this example we discussed the case of a returned pointer assigned to an integer variable, but the same issue happened several times when decompilers declared parameters as integers instead of pointers in the functions prototypes (e.g., BUG-2010 and BUG-2018, that are respectively the UAF and the DF).

Discussion: SAST tools seem to have problems tracking pointers that become integer and later pointer again. Learning from RetDec, the solution is just to back-propagate the type information. In other words, if a variable is later casted to a pointer and de-referenced, then this information should be used to redefine the variable type as a pointer.

For instance, it is sufficient to declare an intermediate variable of type `int*` instead of `v9` in the HexRays's output and all tools that were missing the vulnerability were able to correctly perform their taint analysis until they reach the `memcpy` invocation.

P6 - Integers of Wrong Size

Effect: increase false positives *Repairer: Both*

Decompilers often declare variables of the wrong size (e.g., double-word instead of bytes) and then rely on cast operations to ensure the type system coherency of their output statements. This behavior caused many SAST tools to generate false alarms due to the potentially erroneous pointer casting.

As an example we can consider the code snippets in Listing 12 (original code) and Listing 13 (decompiled code).

```
uint8_t out[SIZE]; uint8_t tmpout[SIZE];
for(j = 0; j < sizeof(out); j++)
  out[j] ^= tmpout[j];
```

Listing 12: Suspicious cast source code

```
__int64 v22;  __int64 v26;
for ( j = 0LL; j <= 0x1F; ++j)
*((_BYTE *)&v22+j) ^= *((_BYTE *)&v26+j);
```

Listing 13: Suspicious cast decompiled code

In the original code, the elements are of type `uint8_t` (i.e., one byte each). In the decompilers output the two variables becomes

64-bit integers, which are later casted to `_BYTE` to perform the xor operation. Furthermore, the retrieval of the *j*-th element is done through pointers arithmetic with type `uint8_t`.

A similar pattern appears very often in our experiments, with different source pointer types and using different types to perform the cast. While this pattern is similar to the *Pointers as Integers* (in fact, again the decompiler used integer variables to store pointers) here is the wrong size and the cast operation that cause false alarms instead of the inability to follow the data flow as in the previous pattern.

It is also interesting to note how, because of the initial declaration of the variables representing the arrays (`__int64 v22` and `__int64 v26` are integer types and not integer pointers), the pattern is reported by some SAST tools as a dangerous cast, rather than a buffer overflow.

On the other hand, if in the previous code, the two variables were defined as `__int64*` we would still observe an alert warning for a potentially unsafe memory access, converging in the case described for **P1**.

Discussion:

This is again a case of type confusion, less severe than the pointer case (as it cannot lead to missing real vulnerabilities), but somehow harder to fix. In fact, back-propagating information to mark variables as pointers is not sufficient, and correctly sizing all integers requires a more complex analysis and inference techniques.

P7 - Simplified Expressions

Effect: decrease detection rate *Repairer: SAST*

This last pattern is quite unusual, but we report it as it is the cause of some missed vulnerabilities in the decompiled code. For our discussion we use the CVE-2017-1000249, a stack BOF present in the *file* project. The original source code is depicted in the following snippet:

```
do_bid_note(.., unsigned char* nbuf, ..)
{
if (namesz == 4 && ... &&
    type == NT_GNU_BUILD_ID &&
    (descsz >= 4 || descsz <= 20)) {
      uint8_t desc[20];
      memcpy(desc, &nbuf[doff], descsz); } ..}
```

Listing 14: Source code of the BOF

The `memcpy` is unsafe because of the wrong check that is performed before its invocation. In fact, the `OR` operator is used instead of the `AND` to check if the size (`descsz`) is in the appropriate range. The boolean condition always evaluates to True, and this is detected by some tools (such as CPPCheck) and reported as potential bug—which in this case it is and leads to a buffer overflow.

However, compilers are also able to detect that the condition is always satisfied and they can simplify the code accordingly. This results in the following decompiled code:

```
vulnerable_foo(..., int a4, ...) {

char v58;
if (v49 == 4 && ... &&
    v28 == 3) {
  memcpy(&v58, a4 + v45, v16);
} }
```

Listing 15: Decompiled code of the BOF

The `desc` buffer is another example of the inability of decompilers to reconstruct stack-based arrays. But the key element for this pattern is that the wrong test on the buffer size is not present anymore. Since the compiler is not generating its corresponding assembly code in the first place, the decompilers have no way to recover it.

6 DISCUSSION AND CONCLUSIONS

We can distill the findings of our experiments around four main points.

(1) The main impediment to the use of SAST tools on pseudocode is that the decompiled code cannot be re-compiled out-of-the-box. The recent paper by Schulte et al. [57] make us feel optimistic that this problem will soon be solved. However, so far, human analysts need to manually fix the decompiled code, a process that becomes prohibitively complex for large codebases.
(2) Once the re-compilable issue is solved, existing SAST tools can discover (in our experiments) 71% of vulnerabilities they were finding in the original code. While there is still a margin for improvement, this result already goes beyond our initial expectations. On the negative side, the number of false positives often increased considerably, making the output of many tools difficult and time-consuming to navigate. However, even if the FP increase is on average 232%, in 29/61 cases the FPs either decreased or did not significantly increase.
(3) Both the compiler and decompiler transformations contribute to the final result in a complex way. Our experiments show that there is no linear trend, and in some cases more aggressive optimizations even simplified the job of the SAST tools.
(4) Today, decompilers are still designed to generate code that is easy to understand for humans, and SAST tools are still designed to parse "well-written" code that is not generated by a machine. This human-centric view could, and should, change in the future. In Section 5 we listed 7 root causes that explain the differences we observed in the results. We believe that many of the entries in our list could be solved, or at least mitigated, by improvement in either the decompiler or the SAST analysis (or both).

In summary, our case studies show that we are approaching the point of convergence of source and binary analysis. While few obstacles still remain, we believe that future work will be able to overcome these issues focusing on both the decompilation side and the static analysis part.

ACKNOWLEDGEMENTS

This research was partially supported by the Defense Advanced Research Projects Agency (DARPA) under grant agreements FA875019C0003 and N6600120C4020.

REFERENCES

[1] Accessed December 4, 2021. Avast Retargetable Decompiler IDA Plugin. https://blog.fpmurphy.com/2017/12/avast-retargetable-decompiler-ida-plugin.html.
[2] Accessed December 4, 2021. Awesome Static Analysis. https://github.com/analysis-tools-dev/static-analysis.
[3] Accessed December 4, 2021. C and C++ Source Code Analysis Tools. https://www.codeanalysistools.com/?cplusplus.
[4] Accessed December 4, 2021. Code-QL. https://securitylab.github.com/tools/codeql.
[5] Accessed December 4, 2021. Code-ql queries examples. https://help.semmle.com/QL/learn-ql/cpp/ql-for-cpp.html.
[6] Accessed December 4, 2021. CPPCheck. http://cppcheck.sourceforge.net/.
[7] Accessed December 4, 2021. CWE Checker. https://github.com/fkie-cat/cwe-checker.
[8] Accessed December 4, 2021. flawfinder. https://github.com/david-a-wheeler/flawfinder.
[9] Accessed December 4, 2021. framac. https://frama-c.com/.
[10] Accessed December 4, 2021. Ghidra. https://ghidra-sre.org/.
[11] Accessed December 4, 2021. Hex-Rays Decompiler. https://www.hex-rays.com/products/decompiler/.
[12] Accessed December 4, 2021. IKOS. https://github.com/NASA-SW-VnV/ikos.
[13] Accessed December 4, 2021. Infer. https://fbinfer.com/.
[14] Accessed December 4, 2021. Joern. https://joern.io/.
[15] Accessed December 4, 2021. Joern queries examples. https://github.com/ShiftLeftSecurity/joern/tree/master/joern-cli/src/main/resources/scripts/c.
[16] Accessed December 4, 2021. RATS. https://code.google.com/archive/p/rough-auditing-tool-for-security/.
[17] Accessed December 4, 2021. Scan-build. https://clang-analyzer.llvm.org/.
[18] Accessed December 4, 2021. Veracode. https://www.veracode.com/products/binary-static-analysis-sast.
[19] Accessed December 4, 2021. What are the best sast tools? https://cybersecuritykings.com/2020/02/16/11-tips-on-sast-tool-selection/.
[20] H.H. AlBreiki and Q.H. Mahmoud. 2014. Evaluation of static analysis tools for software security. In *IIT*.
[21] A. Arusoaie, S. C., V. Craciun, D. Gavrilut, and D. Lucanu. 2017. A comparison of open-source static analysis tools for vulnerability detection in c/c++ code. In *IEEE SYNASC*.
[22] Dirk Beyer and M Erkan Keremoglu. [n. d.]. CPAchecker: A tool for configurable software verification. In *International Conference on Computer Aided Verification*.
[23] D. Brumley, J. Lee, E.J. Schwartz, and M. Woo. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In {*USENIX*}.
[24] G. Chatzieleftheriou and P. Katsaros. 2011. Test-driving static analysis tools in search of C code vulnerabilities. In *IEEE COMPSAC*.
[25] B. Chess and G. McGraw. [n. d.]. Static analysis for security. *2004 IEEE S&P* ([n. d.]).
[26] C. Cifuentes and K. J. Gough. [n. d.]. Decompilation of binary programs. *Software: Practice and Experience* ([n. d.]).
[27] Y. David, N. Partush, and E. Yahav. 2018. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices* (2018).
[28] A. Dinaburg and A. Ruef. 2014. Mcsema: Static translation of x86 instructions to llvm. In *ReCon*.
[29] E.N. Dolgova and A.V. Chernov. 2009. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software* (2009).
[30] P. Emanuelsson and U. Nilsson. 2008. A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science* (2008).
[31] A. Fatima, S. Bibi, and R. Hanif. 2018. Comparative study on static code analysis tools for c/c++. In *IEEE IBCAST*.
[32] J. Feist, L. Mounier, S. Bardin, R. David, and M. Potet. 2019. Finding the needle in the heap: combining static analysis and dynamic symbolic execution to trigger use-after-free. In *SSPREW*.
[33] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao. 2019. Coda: An end-to-end neural program decompiler. In *Advances in Neural Information Processing Systems*. 3708–3719.
[34] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and David V. 2003. Buffer overrun detection using linear programming and static analysis. In *ACM CCS*.
[35] A. Gussoni, A. Di Federico, P. Fezzardi, and G. Agosta. 2020. A Comb for Decompiled C Code. In *ACM AsiaCCS*.
[36] B. Hackett, M. Das, D. Wang, and Z. Yang. 2006. Modular checking for buffer overflows in the large. In *ICSE*.
[37] D. Hovemeyer and W. Pugh. 2007. Finding more null pointer bugs, but not too many. In *ACM SIGPLAN-SIGSOFT PASTE*.
[38] D. Hovemeyer, J. Spacco, and W. Pugh. 2005. Evaluating and tuning a static analysis to find null pointer bugs. In *ACM SIGPLAN-SIGSOFT PASTE*.
[39] D. S. Katz, J. Ruchti, and E. Schulte. 2018. Using recurrent neural networks for decompilation. In *IEEE SANER*.
[40] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav. 2019. Towards neural decompilation. *arXiv preprint arXiv:1905.08325* (2019).
[41] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim. 2020. FirmAE: Towards Large-Scale Emulation of IoT Firmware for Dynamic Analysis. In *ACSAC*.
[42] Y. Kim, J. Lee, H. Han, and K. Choe. 2010. Filtering false alarms of buffer overflow analysis using SMT solvers. *Information and Software Technology* (2010).
[43] K.J. Kratkiewicz. 2005. *Evaluating static analysis tools for detecting buffer overflows in c code.* Technical Report. HARVARD UNIV CAMBRIDGE MA.
[44] J. Křoustek, P. Matula, and P. Zemek. 2017. Retdec: An open-source machine-code decompiler.
[45] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *IEEE/ACM ASE*.
[46] J. Lee, T. Avgerinos, and D. Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).
[47] H. Liang, S. Liu, Y. Zhang, and M. Wang. 2017. Improving the precision of static analysis: Symbolic execution based on GCC abstract syntax tree. In *SNPD*.
[48] Z. Liu and S. Wang. 2020. How far we have come: testing decompilation correctness of C decompilers. In *SIGSOFT ISSTA*.
[49] Z. Liu, Y. Yuan, S. Wang, and Y. Bao. 2022. SoK: Demystifying Binary Lifters Through the Lens of Downstream Applications. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 453–472. https://doi.org/10.1109/SP46214.2022.00027
[50] S. Ma, M. Jiao, S. Zhang, W. Zhao, and D.W. Wang. 2015. Practical null pointer dereference detection via value-dependence analysis. In *IEEE ISSREW*.
[51] R. Mahmood and Q.H. Mahmoud. 2018. Evaluation of static analysis tools for finding vulnerabilities in Java and C/C++ source code. *arXiv preprint arXiv:1805.09040* (2018).
[52] R. K McLean. 2012. Comparing static security analysis tools using open source software. In *IEEE SERE*.
[53] M. Noonan, A. Loginov, and D. Cok. 2016. Polymorphic type inference for machine code. In *ACM SIGPLAN PLDI*.
[54] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. 2015. Cross-architecture bug search in binary executables. In *IEEE S&P*.
[55] S. Poeplau and A. Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In {*USENIX*}.
[56] D. Pozza, R. Sisto, L. Durante, and A. Valenzano. 2006. Comparing lexical analysis tools for buffer overflow detection in network software. In *COMSWARE*.
[57] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov. 2018. Evolving exact decompilation. In *BAR*.
[58] S. Shiraishi, V. Mohan, and H. Marimuthu. 2015. Test suites for benchmarks of static analysis tools. In *IEEE ISSREW*.
[59] E. Söderberg, T. Ekman, G. Hedin, and E. Magnusson. 2013. Extensible intraprocedural flow analysis at the abstract syntax tree level. *Science of Computer Programming* (2013).
[60] J. Viega, J. Bloch, Y. Kohno, and G. McGraw. [n. d.]. ITS4: A static vulnerability scanner for C and C++ code. In *2000 ACSAC*.
[61] D. A. Wagner, J. S Foster, E. A. Brewer, and A. Aiken. 2000. A first step towards automated detection of buffer overrun vulnerabilities.. In *NDSS*.
[62] R. Xu, P. Godefroid, and R. Majumdar. 2008. Testing for buffer overflows with length abstraction. In *ISSTA*.
[63] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. 2016. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *IEEE S&P*.
[64] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations.. In *NDSS*.
[65] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. [n. d.]. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE S&P*.
[66] F. Yamaguchi, M. Lottmann, and K. Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *ACSAC*.
[67] H. Yan, Y. Sui, S. Chen, and J. Xue. 2017. Machine-learning-guided typestate analysis for static use-after-free detection. In *ACSAC*.
[68] H. Yan, Y. Sui, S. Chen, and J. Xue. 2018. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *ICSE*.
[69] J. Ye, C. Zhang, and X. Han. 2014. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities. In *ACM CCS*.