Mining for Mutation Operators for Reduction of Information Flow Control Violations

Ilya Kosorukov
University College London
London, UK

Daniel Blackwell
University College London
London, UK
daniel.blackwell.14@ucl.ac.uk

David Clark
University College London
London, UK
david.clark@ucl.ac.uk

Myra B. Cohen
Iowa State University
Ames, IA , USA
mcohen@iastate.edu

Justyna Petke
University College London
London, UK
j.petke@ucl.ac.uk

Abstract

The unintentional flow of confidential data to unauthorised users is a serious software security vulnerability. Detection and repair of such errors is a non-trivial task that has been worked on by the security community for many years. More recently, dynamic approaches, such as HyperGI, have been introduced that use hypertesting and genetic improvement to not only detect, but also provide a patch that reduces such information flow control violations. However, empirical studies performed so far have used mostly generic mutation operators, potentially limiting the strength of this approach. In this new ideas paper we mine the National Vulnerabilities Database to find repairs of information leaks. Of 636 issues initially identified, we found 73 fixes that relate to information leaks and come with open source patches to the code. From these, we identified 10 types of mutation operators with potential to fix such issues. Six of these have so far never been used to fix information leaks via automated mutation to the code. We propose that these could help improve effectiveness of tools using the HyperGI approach.

CCS Concepts

• **Security and privacy → Software security engineering**; • **Software and its engineering → Search-based software engineering**.

Keywords

Information Leak Reduction, Information Leak Repair, Genetic Improvement, HyperGI

ACM Reference Format:
Ilya Kosorukov, Daniel Blackwell, David Clark, Myra B. Cohen, and Justyna Petke. 2024. Mining for Mutation Operators for Reduction of Information Flow Control Violations. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3691620.3695308

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695308

1 Introduction

Since the introduction of ChatGPT [22] and other large language models (LLM), researchers have conducted a plethora of empirical studies investigating how well LLMs perform in solving typical software engineering tasks. Sobania et al. [28] conducted one of the earliest studies on the use of ChatGPT for automated program repair (APR), showing that the LLM-prompting approach outperformed several existing APR tools at fixing various functional bugs. However, Steenhoek et al. [29] have shown LLMs perform poorly at repair of software security vulnerabilities. Indeed, we tried ChatGPT as well as a powerful open source model, Llama-3-70B [20], to find and repair an insecure flow of confidential information from uninitialised variables to program output in `atalk` [1]. Neither LLM could identify the bug, let alone fix it[1].

Among software vulnerabilities the problem of leaking confidential information is especially important. It can lead to serious security failures, such as the famous Heartbleed Bug [2]. The verification research community has extensively studied ensuring information flow control (IFC) as part of the programming process [31, 32]. IFC is the problem of ensuring that a software system and a security policy satisfy a security property. As security properties are safety properties, most research into IFC has been via verification tools and static or symbolic analyses [9, 15, 25]. A security policy defines which information can flow between different user groups, and in which direction; typically any flow of information which *violates* the policy is referred to as an *information leak*. Note that as the word leak in the software field is typically associated with *resource leaks* such as *memory leaks*, we have chosen to refer to any *violations* as *insecure flows* to avoid confusion.

Only recently Mesecan et al. [19] proposed HyperGI, an approach that not only detects IFC violations, but also automatically generates patches for such issues using genetic improvement (GI) [24]. Mesecan et al. [18] instantiated the HyperGI approach in a tool called LeakReducer and conducted an empirical study showing that LeakReducer can detect and reduce insecure flows in real-world software. The GI approach within LeakReducer mutates statements in a faulty method, trying to find a patch that passes given sets of tests (which serve as proxies for program behaviour and amount of confidential information leaked[2]). Although LeakReducer was able to automatically detect and fix several insecure flows, its mutation

[1]Responses can be found in Appendix B in our repository [17].
[2]See [18] for details, which we omit here in the interest of space.

set mostly contains generic operators used for regular program repair, that move, copy, or delete code statements. To improve the efficiency and effectiveness for repair of information leaks, we propose to mine existing repairs to form a set of repair patterns.

In this work, we mine the National Vulnerabilities Database (NVD) [21] for repairs of insecure flows. Next, we analyse each repair to see what type of mutation has been used to fix a given vulnerability. Finally, we provide a list of mutation operators which could be integrated into an automated tool, such as LeakReducer, to automatically fix detected IFC violations.

Our search returned 636 bugs, 73 of which were insecure flows with access to open source code repairs. We show that each of the 73 bugs could be fixed by a combination of 10 mutation types, 6 of which have not yet been tried for fixing information leaks using GI. We provide our analysis of all the 73 repairs in our repository [17] to allow for replication and extension of our study. In future work we plan to empirically evaluate whether the derived mutations can indeed improve the current state-of-the-art at information leak reduction using HyperGI.

2 Background & Related Work

Information flow control is the study of ensuring that information flows between users only where allowed by a given security policy [14]. For example, when building an email server a reasonable security policy would state that a user Alice should not be able to learn any information about emails received by another user Bob (unless Alice sent an email to Bob of course) and vice versa. A common non-trivial security policy is that within operating systems; whereby certain information from kernel-space must not be revealed to user-space, such as memory addresses, which can be used in exploits in order to bypass KASLR (kernel address space layout randomisation) [11]. Here, the OS kernel is not a human user, but is nonetheless still a user of the system.

There are a number of ways that information can be revealed to a user; the most obvious being through program output such as a GUI or HTTP response, but also through side-channels such as the execution time [13], memory usage or power consumption [27] for a given operation.

The *non-interference property* was introduced in 1982 by Goguen and Meseguer [12], and states the following: *One group of users, using a certain set of commands, is* non-interfering *with another group of users if what the first group does with those commands has no effect on what the second group of users can see.* While this is defined for 'a set of commands', it can be applied at the level of individual programs or functions within programs too. It is the security policy that defines which sets of users and commands should satisfy the non-interference property.

Attempts at using the dynamic approach of hypertesting [16] (i.e., sets of tests) have been proposed [10, 16, 23] to detect IFC violations. More recently, LeakReducer [18] has been proposed to (semi-) automatically fix violations of the non-interference property. The tool was shown successful in reducing such violations in real-world software, first detecting such leaks using hypertests, and then proposing a patch using the genetic improvement approach.

Genetic improvement (GI) [24] uses automated search to improve existing software. It has been used to improve various functional (e.g., bug fixing) and non-functional (e.g., execution time) software properties. GI searches a space of software patches to find improved software variants. The space is defined by a set of mutation operators. The current implementation of LeakReducer either removes, copies, or inserts an existing code statement. It can also synthesize an `if` or `for` loop condition using variables in existing code. These mutation operators have been inherited from previous work. Here we investigate what types of mutations are in practice applied to code to repair information leaks.

3 Information Leakage Mining

In order to find information leaks and their fixes we turned to the CVE database [30]. The CVE database contains large amounts of publicly disclosed cybersecurity vulnerabilities and is easily searchable. Information leakage errors very commonly become cybersecurity vulnerabilities due to their nature of disclosing sensitive data, therefore we expected to find a satisfactory amount of usable samples in this database. For a code sample to be usable in our project it must meet two criteria:
(1) The program must contain vulnerabilities that were specifically information leakage errors;
(2) The original vulnerable code in the CVE entry is open source, along with a patch that repairs the vulnerability.

While searching for other metadata we could use to determine whether a CVE entry was open source along with a patch provided, or not, we discovered that CVE entries were also present in another database, the National Vulnerabilities Database (NVD) [21]. Unlike the CVE database, the NVD included better maintained metadata tags, including those that label reference URLs. Therefore, we used the NVD API and keywords "leakage" and "information disclosure" to gather relevant data. For each entry we looked for the "Patch" tag that specified whether a given entry contains source code or not. Furthermore, we decided to limit our results to 2020-2022 since this would give us more recent issues that people were experiencing with information leakage, which would make sure that the mutation operator we devise would be applicable to issues that are current. At the same time, we observed later issues rarely contained associated source code, thus we decided to limit our searches to the aforementioned data range for a representative sample. As noted earlier, the term *leak* often appears in the context of resource leaks such as *memory* or *file descriptor* leaks, which not always lead to the disclosure of confidential information.

4 Results

Our mining yielded 636 results. However, many of these results included closed source projects that we could not analyse. After manually filtering out the closed sourced patches we reached 73 results, which allowed us to devise our proposals for new mutation operators. Full results can be seen in our repository [17].

The aforementioned 73 results were analysed and labelled with: the language that the sample is from; whether it is possible for the current GI mutation operators to create the fix; and, if not, what components are there to the patch that could be turned into new mutation operators. Figure 1 shows a breakdown of the count of leaks by programming language. They range from C/C++ to Go, PHP, Python, Javascript, Ruby and a few others such as Rust.

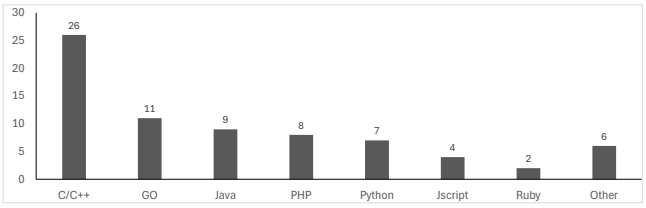

Figure 1: Language distribution of the CVEs studied.

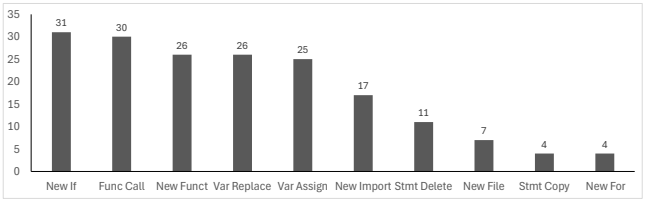

Figure 2: The number of instances found of each mutation type in the 73 mined fixes to information leaks.

Figure 2 shows how many instances of mutation types we found throughout our 73 collected results. By mutation type we mean a type of edit to the original code. An instance of a mutation type is counted when it is present any number of times within the fix to the vulnerability. 54 vulnerabilities required more than one type of edit to produce a fix. We found 10 types of mutations, 4 of which (New If, Statement Delete, Statement Copy and New For) were already present in LeakReducer[18], leaving the 6 new mutation types which are described in Section 4.1.

Table 1 shows a sample of our complete data that contains 2 entries. Figure 2 shows a summary of the types of fixes. New If was the most common, appearing 31 times, while a Function Call was the second with 30. We also saw Variable Assignments, Variable Replacement, New Functions and New Import statements.

4.1 New Types of Mutations

We classified and grouped patches by manual observation. With the main criteria being the possibility of the mutation types within those patches to be turned into a mutation operator.

4.1.1 Variable Replacement. This is a mutation where one variable name gets replaced with another variable. Information flow can be easily affected by this type of mutation. In the example of a simple assignment from one variable to another, confidential information can be accidentally transmitted to an unclassified variable, accessible by unauthorised users.

The example in Listing 1 is part of a series of patches fixing an information leakage bug. The original issue was caused by a buffer over-read. To solve this issue as shown in Listing 1, the buffer access offset was corrected.

Listing 1: Patch for bug CVE-2022-0891 [5]. Variable marked in red is replaced with the code snippet in green.

```
bitset=*(src_buff + offset2) & (((unsigned char)1<<k)) ? 1 : 0;
bitset=*(src_buff + offset1 + full_bytes)
       & (((unsigned char)1<<k)) ? 1 : 0;
```

Table 1: Sample of complete data that contains 2 entries which show types of mutations in analysed patches. Full data can be found in Appendix A in our repository [17].

- CVE ID: This field contains the CVE ID of the investigated vulnerability.
- Patch: This field contains a URL that leads to the patch or series of patches that fixed the vulnerability.
- Language: This field contains the language that the vulnerable software was written in.
- Possible: This field contains whether it would be theoretically possible to solve this vulnerability with the mutation operators currently present in LeakReducer at time of writing this paper.
- Mutation types: This is a series of fields each titled by a type of mutation that was observed throughout all the data. Each field would specify whether the patch is present within the fix that solved the vulnerability.

CVE ID	CVE-2021-22929	CVE-2022-29567
Patch	hackerone.co...	github.com/...
Language	C++	Java
Possible	No	No
Statement Copy	1	
Statement Delete	1	
New If	1	
New For	1	
Variable Replacement	1	
Function Call	1	1
Variable Assignment		1
New Function	1	1
New File		1
New Import		1

4.1.2 Variable Assignment. This class of change covers errors which were fixed by assigning a value to a pre-existing variable. As mentioned previously, confidential information held in variables can easily be leaked through assignments and this class of fix aims to prevent that from happening by being able to adjust the variable.

In Listing 2 information is being leaked when the struct `minfo6` is output – in this case copied from the kernel-space `minfo6` to the user-space buffer `iter`. More specifically it is the `flags` field inside the struct that contained sensitive information. To fix this issue the flags field was set to 0 ('*zeroed*') before the struct gets copied, which is shown in Listing 3.

Listing 2: Code from CVE-2019-16714[3] before repair.

```
    minfo6.fport = inc->i_hdr.h_dport; }

rds_info_copy(iter, &minfo6, sizeof(minfo6));
```

Listing 3: Code from CVE-2019-16714[3] after repair.

```
    minfo6.fport = inc->i_hdr.h_dport; }
minfo6.flags = 0;
rds_info_copy(iter, &minfo6, sizeof(minfo6));
```

4.1.3 Function Call. This category of fix describes the cases where a function call is inserted to alter variables or perform a bounds check. Information can easily be leaked by incorrect management of memory, especially in C and C++, therefore a large proportion of errors were simply fixed by inserting function calls to `memset`. Listing 4 presents one such example.

Listing 4: Code from CVE-2022-40768 [8] after repair. `memset` function call added.

```
# The passthrough structure is declared off of the stack, so it needs
# to be zeroed out before copied back to userspace to prevent any un-
# intentional data leakage.
...
        struct st_drvver ver;
        size_t cp_len = sizeof(ver);

        memset(&ver, 0x00, sizeof(ver));
        ver.major = ST_VER_MAJOR;
```

4.1.4 New Import. Many issues required the ability to insert a new import statement. This could be due to the fact a new file was created to address the issue, or a new library is necessary. An example is given in Listing 5.

Listing 5: Python code showing the fix to the vulnerability in CVE-2021-28861[4] involving the import of the `re` (regular expression) module and later the use of its' `sub` (substitution) function.

```
import sys
import re
import time
...
      # bpo-43223: The purpose of replacing '//' with '/' is to
      # protect against open redirect attacks reside within
      # http.server module which can be triggered if the path
      # contains '//' at the beginning because web clients treat
      # /path as an absolute url without scheme (similar to
      # http:/ /path) rather than a relative path
      self.path = re.sub(r'^(/)+', '/', self.path)
```

4.1.5 New Function. When larger changes to the codebase were required, functions were generally created to facilitate these changes. Although this is not necessary to solve the bug it creates a much more elegant solution in the long run, and is therefore observed to be implemented a large number of times.

The solution for CVE-2022-23318 [6] shows one such new function. Here, a function was created for the purpose of bounds checking on a variable, this function was used in multiple locations in the code to solve an information leakage issue.

4.1.6 New File. Similarly, instead of a new function, a whole new file might be created in cases where a larger number of changes are required. This is prevalent in Java or Python code where each class is commonly separated into its own file.

The vulnerability in CVE-2022-39310 [7] stemmed from an issue where universally unique identifiers (UUIDs) would not be properly verified; allowing an authenticated user to act on behalf of another agent which would give them access to classified information. The issue was solved by introducing a new custom class to handle the authentication process, ensuring proper verification of the user's UUID. This new class was placed into a new file.

5 Discussion

We identified 6 new types of mutations that have not yet been implemented for automated fixing of information leaks. We observed that certain types of mutations are more common for particular programming languages, e.g., Function Call and New If for C/C++ vs New Import and Variable Replace for Java. However, our sample is too small to draw general conclusions. Exploring how to best integrate these new mutation types into existing secure development practices could bring new insights. Here we discuss the possibility of incorporating these in a GI-based tool, i.e., LeakReducer [18].

Variable Replacement Variable Replacement is one of the most commonly occurring mutation types. It is also easy to implement, as the only requirements are to know what variables exist in the source code within a given scope, and at which locations they can be inserted. Both of these requirements can be easily implemented into LeakReducer as the tool internally uses an XML representation of code, with nodes tagged according to parse type, thus a node of type "variable" can easily be identified.

Variable Assignment An implementation of this mutation operator would introduce some complexity in LeakReducer. This operator is able to, in theory, assign any value to any existing variable. Although the value being assigned can often be narrowed down to the correct type, it cannot be narrowed down to the value that should be assigned. Using random values until a suitable value is found is likely to take an unreasonable amount of time. Nevertheless, for variables with small ranges of values such an operator could potentially be added. One could also scrape constant values (including const's and MACROS for C/C++) from the code.

Function Call This type of mutation operator would be very useful for information leakage errors that have to do with, for example, zeroing memory. Not only could it change when or if memory is zeroed but this mutation operator would also be able to change how much memory is zeroed, due to its ability to mix and match arguments. This operator could be added to LeakReducer, as well as the aforementioned *Variable Replacement* operator. Implementing a function call requires only analysis of available functions and variables in the source code and a location for insertion.

New Function & New Import & New File LeakReducer currently makes changes within a single file. To implement operators involving significant amounts of new code, one would have to specify, for instance, which libraries could be used to import, or be able to synthesize new functions and classes. Due to the large search space, it would be currently impractical to implement such operators. Although techniques such as genetic programming [26] or LLMs could synthesize new code, defining the desired behaviour and where it should be inserted poses a significant challenge.

6 Conclusions

The problem of repairing information leaks is non-trivial and has been researched by the security community for many years. Recently a dynamic approach, HyperGI [19], has been introduced to detect and repair information leak issues. Its instantiation in LeakReducer [18] mostly uses generic mutation operators. In this work we mined a popular software vulnerability database to see how such issues are fixed in practice. We identified 73 relevant bugs with open source code. Each of them can be fixed by applying a combination of 10 types of mutation operators, 6 of which are missing from LeakReaducer. We provide analysis of the 73 bug repairs in our repository [17]. In the future we plan to extend LeakReducer with the identified mutation types to test if the new mutations indeed increase the tool's effectiveness at fixing information leaks.

Acknowledgments

Supported by grants: NSF #1909688, UK EPSRC #EP/S022503/1.

References

[1] 2009. CVE-2009-3002. Available from MITRE, CVE-ID CVE-2009-3002. https://www.cve.org/CVERecord?id=CVE-2009-3002

[2] 2014. CVE-2014-0160. Available from MITRE, CVE-ID CVE-2014-0160. https://www.cve.org/CVERecord?id=CVE-2014-0160

[3] 2019. CVE-2019-16714. Available from MITRE, CVE-ID CVE-2019-16714. https://www.cve.org/CVERecord?id=CVE-2019-16714

[4] 2021. CVE-2021-28861. Available from MITRE, CVE-ID CVE-2021-28861. https://www.cve.org/CVERecord?id=CVE-2021-28861

[5] 2022. CVE-2022-0891. Available from MITRE, CVE-ID CVE-2022-0891. https://www.cve.org/CVERecord?id=CVE-2022-0891

[6] 2022. CVE-2022-23318. Available from MITRE, CVE-ID CVE-2022-23318. https://www.cve.org/CVERecord?id=CVE-2022-23318

[7] 2022. CVE-2022-39310. Available from MITRE, CVE-ID CVE-2022-39310. https://www.cve.org/CVERecord?id=CVE-2022-39310

[8] 2022. CVE-2022-40768. Available CVE-ID CVE-2022-40768. https://www.cve.org/CVERecord?id=CVE-2022-40768

[9] Fabrizio Biondi, Axel Legay, Louis-Marie Traonouez, and Andrzej Wasowski. 2013. QUAIL: A Quantitative Security Analyzer for Imperative Code. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8044)*, Natasha Sharygina and Helmut Veith (Eds.). Springer, 702–707. https://doi.org/10.1007/978-3-642-39799-8_49

[10] Daniel Blackwell, Ingolf Becker, and David Clark. 2023. Hyperfuzzing: black-box security hypertesting with a grey-box fuzzer. arXiv:2308.09081 [cs.SE] https://arxiv.org/abs/2308.09081

[11] Haehyun Cho, Jinbum Park, Joonwon Kang, Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, Adam Doupé, and Gail-Joon Ahn. 2020. Exploiting uses of uninitialized stack variables in linux kernels to leak kernel pointers. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[12] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*. 11–11. https://doi.org/10.1109/SP.1982.10014

[13] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: Speculative symbolic execution for cache timing leak detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1235–1247.

[14] Daniel Hedin and Andrei Sabelfeld. 2012. A perspective on information-flow control. In *Software safety and security*. IOS Press, 319–347.

[15] Jonathan Heusser and Pasquale Malacaria. 2010. Quantifying information leaks in software. In *Twenty-Sixth Annual Computer Security Applications Conference, ACSAC 2010, Austin, Texas, USA, 6-10 December 2010*, Carrie Gates, Michael Franz, and John P. McDermott (Eds.). ACM, 261–269. https://doi.org/10.1145/1920261.1920300

[16] Johannes Kinder. 2015. Hypertesting: The Case for Automated Testing of Hyperproperties. In *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot 2015)*. 1–8. 3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot 2015) ; Conference date: 18-04-2015.

[17] Ilya Kosorukov, Daniel Blackwell, David Clark, Myra B. Cohen, and Justyna Petke. 2024. Our repository with analysis of 73 bug repairs and responses from ChatGPT and LLama-3 when prompted to fix an example memory leak. https://github.com/SOLAR-group/infoleakmining

[18] Ibrahim Mesecan, Daniel Blackwell, David Clark, Myra Cohen, and Justyna Petke. 2023. Keeping Secrets: Multi-objective Genetic Improvement for Detecting and Reducing Information Leakage. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 61, 12 pages. https://doi.org/10.1145/3551349.3556947

[19] Ibrahim Mesecan, Daniel Blackwell, David Clark, Myra B. Cohen, and Justyna Petke. 2021. HyperGI: Automated Detection and Repair of Information Flow Leakage. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 1358–1362. https://doi.org/10.1109/ASE51524.2021.9678758

[20] Meta. [n. d.]. Llama-3-70B. https://huggingface.co/meta-llama/Meta-Llama-3-70B-Instruct Accessed: 14/06/2024.

[21] National Institute of Standards and Technology. [n. d.]. National Vulnerabilities Database. https://nvd.nist.gov/ Last accessed: 14/06/2024.

[22] OpenAI. [n. d.]. ChatGPT. https://openai.com/index/chatgpt/ Accessed: 14/06/2024.

[23] Michele Pasqua, Mariano Ceccato, and Paolo Tonella. 2024. Hypertesting of Programs: Theoretical Foundation and Automated Test Generation. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 115:1–115:12. https://doi.org/10.1145/3597503.3640323

[24] Justyna Petke, Saemundur O. Haraldsson, Mark Harman, William B. Langdon, David R. White, and John R. Woodward. 2018. Genetic Improvement of Software: A Comprehensive Survey. *IEEE Transactions on Evolutionary Computation* 22, 3 (2018), 415–432. https://doi.org/10.1109/TEVC.2017.2693219

[25] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Pasareanu. 2012. Symbolic quantitative information flow. *ACM SIGSOFT Softw. Eng. Notes* 37, 6 (2012), 1–5. https://doi.org/10.1145/2382756.2382791

[26] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A field guide to genetic programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk. http://www.gp-field-guide.org.uk (With contributions by J. R. Koza).

[27] Mark Randolph and William Diehl. 2020. Power side-channel attack analysis: A review of 20 years of study for the layman. *Cryptography* 4, 2 (2020), 15.

[28] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An Analysis of the Automatic Bug Fixing Performance of ChatGPT. In *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE Computer Society, 23–30. https://doi.org/10.1109/APR59189.2023.00012

[29] Benjamin Steenhoek, Md Mahbubur Rahman, Monoshi Kumar Roy, Mirza Sanjida Alam, Earl T. Barr, and Wei Le. 2024. A Comprehensive Study of the Capabilities of Large Language Models for Vulnerability Detection. *CoRR* abs/2403.17218 (2024). https://doi.org/10.48550/ARXIV.2403.17218 arXiv:2403.17218

[30] The MITRE Corporation. [n. d.]. The CVE Database. https://cve.mitre.org/ Last accessed: 14/06/2024.

[31] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kici, Ranjit Jhala, Dean M. Tullsen, and Deian Stefan. 2021. Automatically eliminating speculative leaks from cryptographic code with blade. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434330

[32] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2/3 (1996), 167–188. https://doi.org/10.3233/JCS-1996-42-304