Article

Bitcoin's APIs in Open-Source Projects: Security Usability Evaluation †

Philipp Tschannen [1,*,‡] and Ali Ahmed [2,*,‡]

1   Department of Computer Science, University of Liverpool, Liverpool L69 7ZX, UK
2   School of Engineering and Computer Science, Victoria University of Wellington,
     Wellington 6140, New Zealand
*   Correspondence: philipp.tschannen@online.liverpool.ac.uk (P.T.); ali.ahmed@vuw.ac.nz (A.A.)
†   This paper is an extended version of our paper published in Evaluation and Assessment in Software
     Engineering (EASE2020).
‡   This author contributed equally to this work.

check for
updates

Abstract: Given the current state of software development, it does not seem that we are nowhere near vulnerability-free software applications, due to many reasons, and software developers are one of them. Insecure coding practices, the complexity of the task in hand, and usability issues, amongst other reasons, make it hard on software developers to maintain secure code. When it comes to cryptographic currencies, the need for assuring security is inevitable. For example, Bitcoin is a peer-to-peer software system that is primarily used as digital money. There exist many software libraries supporting various programming languages that allow access to the Bitcoin system via an Application Programming Interface (API). APIs that are inappropriately used would lead to security vulnerabilities, which are hard to discover, resulting in many zero-day exploits. Making APIs usable is, therefore, an essential aspect related to the quality and robustness of the software. This paper surveys the general academic literature concerning API usability and usable security. Furthermore, it evaluates the API usability of Libbitcoin, a well-known C++ implementation of the Bitcoin system, and assesses how the findings of this evaluation could affect the applications that use Libbitcoin. For that purpose, the paper proposes two static analysis tools to further investigate the use of Libbitcoin APIs in open-source projects from a security usability perspective. The findings of this research have improved Libbitcoin in many places, as will be shown in this paper.

Keywords: API usability; Bitcoin; security; privacy; open-source; Libbitcoin APIs; software developers

1. Introduction

The success of Bitcoin as an alternative way of paying money online sparked considerable interest and research in the area of Blockchain, with more interest in Bitcoin's most prominent technologies, such as the proof-of-work scheme. Most of the existing Bitcoin research revolves around the areas of security, privacy and resource usage, as depicted in Figure 1. While some research has been done on the usability of Bitcoin's applications from an end-user's point of view, as far as the authors of this paper know, there is no research yet that addresses usability aspects from a developer's point of view. From a developer's point of view, Bitcoin is a software system that implements digital money. A software system's functionality can be made available to other software systems using Application Programming Interfaces (APIs). Thus, software developers want to create an application that integrates or utilises Bitcoin and can use a software library that provides Bitcoin functionality via such an API. Bitcoin Applications comprehensively deal with monetary assets. A malfunctioned Bitcoin application

consequently puts its users at risk of losing money. It is, therefore, critical that such APIs are easy and safe to use. An essential aspect of APIs' safety is their usability, which is the main focus of this research paper. It is worth emphasising that security verification of the Bitcoin technology or the question regarding the safety of the implementation of Bitcoin's APIs is outside the scope of this research.

Usability is defined by the ISO 9241-11 standard as "the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [1]. This paper focuses on the usability of security in the context of Bitcoin's APIs. The users of APIs are software developers. Consequently, from a software developer's perspective, APIs, for which security is of a major concern, should be designed in such a way that makes it difficult, if not impossible, for client code to introduce security vulnerabilities by misusing an API. Unfortunately, this is not the general case, as discussed by Acar et al. (2017) in [2]. The incorrect use of APIs may result in critical security vulnerabilities, as demonstrated by Myers et al. (2016) in [3]. In the software industry, it is not surprising to find APIs that are poorly documented, hard to learn, inconsistent, or facilitating bugs in the code, which diminishes the benefits gained from using such APIs, as discussed by Zibran et al. (2011) in [4]. Acar et al. (2016) in [5] highlighted that usable security still seems to be a secondary concern for developers. Aspects such as functionality, time-to-market, economics and compliance with corporate policies all seem to have greater importance than making software or APIs secure to use.
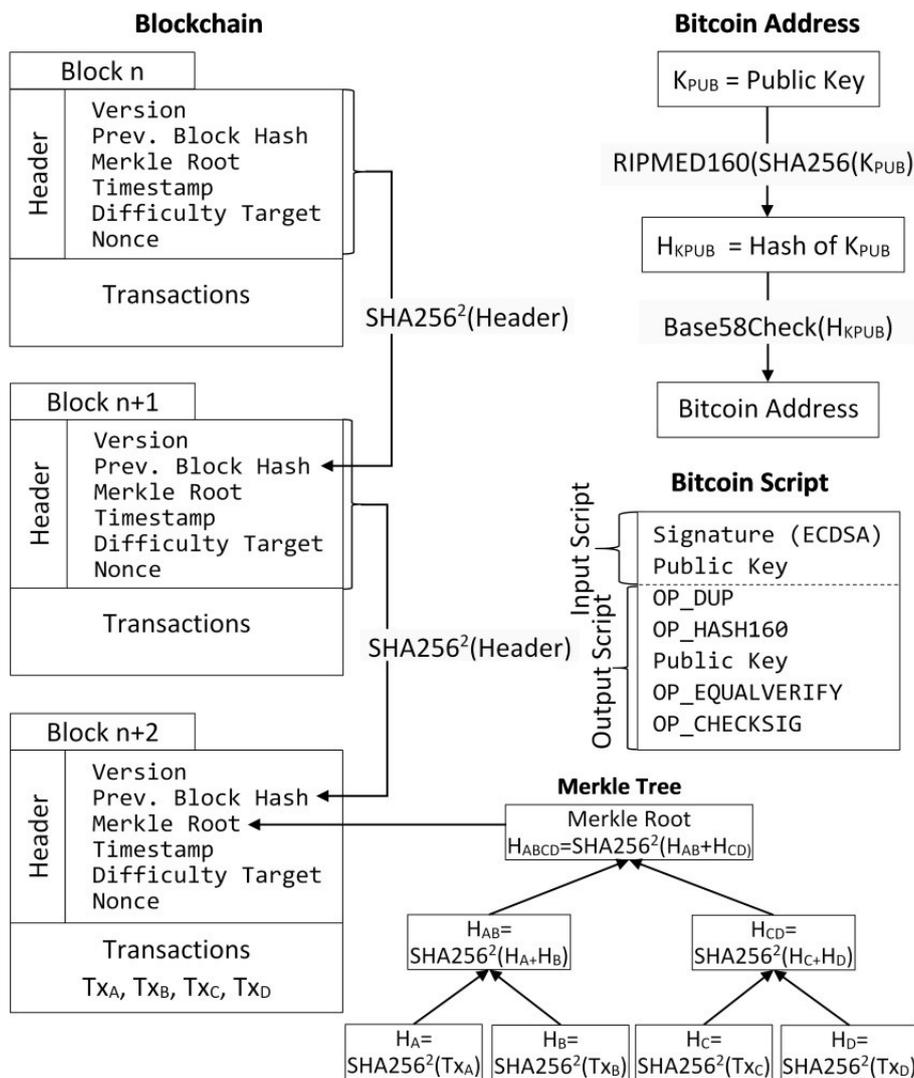

Figure 1. Bitcoin's cryptographic concepts.

What differentiates Bitcoin libraries from general-purpose libraries is that for Bitcoin libraries, security is paramount. Typically, misusing a general-purpose library results in program misbehaviour. While such misbehaviour can sometimes be severe, often it is just annoying. With security APIs, however, misuse can result in misbehaviour that is critical or costly. With Bitcoin libraries, in particular, applications that use such libraries put their users at risk of losing money. The importance of the secure use of Bitcoin APIs, and security APIs in general, shifts the focus of API usability evaluation to some specific first-level attributes of the API usability taxonomy. The first-level usability attribute, knowability, is relevant for any type of API. However, operability, efficiency and subjective satisfaction are evidently less critical for security-related APIs. Instead, robustness and safety become the most crucial aspect of the API usability attributes.

Given the aforementioned challenges, this paper surveys the general academic literature concerning API usability and usable security. It evaluates the API usability of Libbitcoin, a well-known C++ implementation of the Bitcoin system, and assesses how the findings of this evaluation could affect the applications that use Libbitcoin. For such a purpose, the paper proposes two static analysis tools. Besides, this paper studies the use of Bitcoin APIs in open-source projects from a security usability perspective. The findings of this research have improved Libbitcoin in many places, as will be shown in the paper. The organisation of this paper is as follows. Section 2 discusses the research methodology of the paper, demonstrating the data sources, search strategy and how data is extracted along with the inclusion and exclusion criteria. Section 3 surveys the related work. Section 4 introduces the two static analysis tools, including the design objective and the building blocks. Section 5 evaluates the results and lists our observations. Finally, Section 6 concludes the paper and draws the future direction of the work. The novel contribution of this paper focuses on being the first, as to the knowledge of the researchers, work that focuses on the usability of security of the Bitcoin's libraries and their application in open-source projects. It, also, systematically surveys the existing literature regarding usable security from a software developer's point of view. It serves as a road-map of the research efforts in the field of usable security from a software developer's point of view while introducing the key lessons learned from those research proposals.

2. Research Methodology

In this paper, those individual study papers concerning usable security from the developer's point of view are studied. The main sources of data here are ACM DL (https://www.springer.com/gp) accessed 03.03.2020, IEEE Xplore (https://ieeexplore.ieee.org/Xplore/home.jsp) accessed 03.03.2020, and Springer (https://www.springer.com/ accessed 03.03.2020). The Google Scholar (https://scholar.google.com accessed 03.03.2020) service is also used for some aggregated searches. Given those data sources, the searches are narrowed down to the following areas: Security and Privacy, Software Engineering, and Programming Languages. Conferences such as ICSE, FSE/ESEC, ASE, ISTTA, ICSME, POPL, PLDI, OOPSLA, ICFP, IEEE Security & Privacy, ACM CHI, SOUP, and USENIX are comprehensively researched. The set of surveyed papers (i.e., the individual study papers, as defined by Kitchenham in [6]) are believed to offer a sound body of literature for usable security.

This paper follows the same criteria of Merino et al. (2019) in [7] that only reviews the proceedings of the venues with full papers and excludes other types of papers such as tool posters, keynotes and so forth. The criteria we used to include/exclude papers is the date, peer-review status, the type of publication, and the number of citations. For individual study papers, the abstract was firstly read, and then the conclusion was read. In some cases that required further study, the full paper was read. Most of the papers listed in this research were identified/categorised without reading the full paper. To extract the data from those individual study papers, a mix of manual and automated keyword extraction was utilised. Manually, the words found in the keywords section of those papers were collected, if any. When the keyword section was missing, the free online service from FiveFilters organisation (http://termextract.fivefilters.org/extract.php accessed 03.03.2020) was used with $Maxitems = 30$ and $Maxwordsperterm = 3$. Then, the list of the resulting keywords was manually inspected against

the following terms: "Usability", "Usable Security", "Static analysis tool", "IDE tools", "Secure Programming", "secure programming language", "security learning", "security learning resources", "security perception", "APIs usability", "Secure coding", "Secure software development" and "Software developer".

3. Related Work

According to Rama and Kak (2015, p. 76), "an API can be thought of as a focused expression of the overall functionality of a software module in terms of method declarations that can be called by others wishing to utilise the services offered by the module" [8]. More specifically, an API is a means for a software system to provide some of its functionality to other software systems. APIs are typically a set of function declarations that describe the functionality or services offered by a software system. Assurance is needed to assure those APIs are sound from a security perspective. Software security assurance techniques have been around for some time. To assure safety, the developers' use of such techniques differs. The developers' practices to ensure the security of their code range from the use of peer code review to static analysis to penetration testing. Wagner et al. (2015) reported the overhead of such techniques could be too time-consuming, and that is why many developers are not willing to tolerate that in production [9]. Other studies report integration problems, untrustworthiness, overwhelming non-actionable warnings and being too expensive to fix amongst the reasons why developers do not largely use static analysis tools [10]. Generally, the large number of alarms is a major concern with the static analysis tools, as reported by Muske and Serebrenik (2016) in [11]. This is a serious usability problem. Early studies and surveys on API usability, such as McLellan et al. (1998) in [12], Stylos et al. (2008) in [13] and Robillard (2009) in [14] mainly focused on the increased productivity aspect of API usability. Their common finding was that complementary resources, such as documentation and code examples, were instrumental for software developers to grasp and master new APIs quickly. However, considering the grave consequences that security exploits can have, the focus on productivity must be put into context. While producing secure APIs may take a considerable time, this investment will pay off in the long run. The API usability issues can lower code quality and ruin the productivity gains achieved, for example, through reuse. This was argued by Zibran et. al (2011) in [4], where they found that 562 of 1513 bug-posts across five different bug repositories were related to API usability. They tagged all the 562 API usability-related bug-posts with the appropriate usability factors taken from a selection of 22 usability factors that were earlier published by Zibran (2008) in [15]. Finally, they used the study's findings to list factors that affect usability and to determine their significance. A study by Piccioni, Furia and Meyer (2013) in [16], focusing on API documentation and tool support, highlighted the importance of naming to convey the semantics of API functions. They further pointed out the importance of accurate documentation and that it may be more difficult for programmers unfamiliar with the conventions of a particular programming language to intuitively understand the APIs targeting that programming language. Finally, they found that the appreciation of choice (i.e., the flexibility offered by a particular API) differs between novice and experienced software developers. 'Flexibility' and 'appreciation of choice' are, sometimes, the cause of mistakes. For example, password usability does not only affect end-users but also software developers. While end-users are more concerned with the password complexity and how hard it is to break, software developers are more into making sure their applications store passwords securely. Many developers are still storing plain passwords, not the salted-hash value of the passwords.Thus, should the password storage be compromised, the whole system falls apart. Alena et al. (2017) in [17] studied how developers (20 student participants) deal with password storage. The main result, as reported by the study, is that most of the participants claim they have little understanding of the topic of secure password storage. An observation in similar studies that utilise students to reflect the software developers community, raises a concern regarding students instead of professional developers being recruited to investigate a specific industrial practice. How accurate such a practice is, has been investigated by Acar et al. (2016) highlighted in [18]). An information resource such as Stack Overflow

(https://stackoverflow.com/ accessed 03.03.2020) is claimed to be one of the reasons for insecure code by inexperienced developers. Research questions such as "What is the best resource to learn security?" and "What does make an information resource a good one for learning security?" have been around for some time now. Acar et al. (2016) in [18] investigated how information resources impact code security. In other words, how programmers learn security. The findings are interesting when the resulting code is evaluated against functionality and security. The developers using Stack Overflow created significantly better functional code and significantly less secure code compared to those using the Android official documentation, for example. On the same line of work, Fischer et al. (2017) in [19] studied the impact of code 'copy and paste' from Stack Overflow on the code security, which has been studied further by Acar et al. (2017) in [20]. The simple finding in this work is that there exists excessive copying for insecure code snippets from Stack Overflow in current Android applications. About 15.4% copied&pasted code from Stack Overflow with about 97.9% of that containing at least one insecure code snippet. The shocking thing here is that the code snippets copied in those 15.4% applications are security-related code, not just for functional requirements! The work of Imai and Kanaoka (2018) in [21] confirmed that, by studying the actual vulnerable code that is used widely from Stack Overflow. Motivating developers to learn security is a rather challenging task. Weir et al. (2016) call for 'fun security learning' in their work in [22], which shares the same opinion of Boopathi et al. (2015) [23] that fun security learning is an effective method for cyber-security education. The work in [22] suggests the use of techniques from other domains to motivate app developers to learn security. Techniques include games that teach and story-telling. The idea of using games to teach security is interesting. Tillmann et al. (2014) were the first to call for gamifying programming learning in their work in [24]. They proposed Code-Hunt (https://www.microsoft.com/en-us/research/project/code-hunt/ accessed 03.03.2020); a game that teaches general programming skills. Denning et al. (2013), in their work in [25], designed and implemented a card game to teach developers security. Although there were no developer's interviews, the application is distributed to 150 educators who used the app with their students. How much computer science students reflect genuinely the state of professional developers is a question that has been indeed discussed by Acar et al. (2016) in [5]. For those interested in reading about gamifying security training, the suggestion is to read the M.Sc. thesis of Rieff (2018) in [26].

While most of API usability studies focus on a qualitative analysis approach based on some defined guidelines and usability attributes, Rama and Kak (2015) in [8] take a quantitative approach, proposing eight novel metrics of analysing API usability based on nine structural measures. These nine structural measures represent issues with practical relevance and cover topics such as inconsistencies in function definitions, unmanageable function argument lists, failure to correctly group and name functions, problems related to concurrency and exception handling and poor documentation. Green and Smith (2016) in [27] suggest ten principles for designing usable security APIs. Although their focus is on cryptographic APIs, most of their suggested principles apply equally well to other security APIs. Lo Iacono and Gorski (2017) in [28] note that most of the research investigating the APIs' usability of security is related to cryptography. They point out that security APIs include more than cryptographic APIs. Consequently, they suggest a classification scheme to structure the field of security APIs and use that classification scheme in two other studies. The conclusion of this work is that further research is needed for the various types of security APIs. Mosqueira-Rey et al. (2018) in [29] propose a new set of usability heuristics and guidelines, some of which they synthesised and refined from existing literature. The presented heuristics and guidelines form a conclusive classification scheme that can be used to analyse the usability of APIs. After they conducted a study to evaluate their usability taxonomy, Mosqueira-Rey et al. (2018) concluded that the existing literature on API usability, while somewhat complementary, is not entirely complete. They pointed out that quantitative analysis of API usability must be complemented with qualitative analysis, as usability is not entirely objective. That is why Cognitive Walkthrough is, sometimes, used to evaluate certain usability attributes [30].

There exists only little developer-centric literature about Bitcoin, and most of it is not academic. Antonopoulos (2017) in [31] describes how to build Bitcoin Core, discusses the reference Bitcoin

implementation, provides some information about how to use the JSON-RPC API and presents most of Bitcoin's core technical aspects from a developer's point of view. A comprehensive resource about Bitcoin Core development with a detailed description of its technical internals is Bitcoin official website (https://Bitcoin.org/en/ accessed 03.03.2020). Some information about how to use Libbitcoin can also be found in [31]. Additionally, Libbitcoin's GitHub project Wiki provides code examples and other developer's documentation (https://github.com/Libbitcoin/Libbitcoin/wiki accessed 03.03.2020).

According to Yli-Huumo et al. (2016) in [32], there did not exist any research on the usability of Bitcoin from a developer's perspective by 2016, although the unfriendly nature of Bitcoin's API has been noted by Meva (2016) in [33]. Unfortunately, the difficulty of using Bitcoin APIs had not yet been addressed academically. This situation does not seem to have changed since Yli-Huumo et al. (2016) published their study. However, there exists a growing body of research unrelated to Bitcoin that covers aspects related to this paper. Table 1 summarises the major academic pieces of work that are related to the API usability, APIs usable security, API usability heuristics and guidelines, API misuse detection and so forth. As far as the researchers of this paper know, and based on Table 1, there has been no previous work to investigate the security usability of Bitcoin's APIs. We have also categorised those workpieces in Table 2. It is worth noting that Table 2 categorises those related workpieces to the general security usability from a developer's perspective, not just the APIs. The categories are:

1. **Behavioural**, which studies the behavioural practices of the software developers in developing secure code. This includes the following subcategories:

    (a) **Security Learning**, which covers the research done investigating the existing security learning methods and resources available.
    (b) **Perception** that studies what the developers think about secure coding and security in general.
    (c) **Development Processes**, which investigate the available techniques to help developers improve code security.

2. **Programming**, which studies the programming practices of the software developers concerning secure coding. This includes the following subcategories:

    (a) **Languages** that either surveys existing literature or proposing new secure programming languages.
    (b) **Tools** that are used in security assurance mainly to help the software developers writing secure code.
    (c) **Implementation Choices**, which investigates the reasons for software developer's bad coding behaviour and the bad design decisions developers take that affect the security of their applications.

As this research is focused more on the tools category, other categories are outside the scope of this work. Professional software developers usually select a programming language to use on the basis of its fitness for the task at hand. Furthermore, software developers often have certain programming language preferences. Some software developers, for example, prefer to use a language with which they are familiar. Others prefer to use new or hip languages. This project looks at Bitcoin APIs. Unlike other security APIs, such as cryptography APIs, for example (i.e., the workstream of Yasemin Acar, such as the one in [2]), there are not many programming languages with more than one well-maintained Bitcoin implementation. Instead, there are typically only one or two maintained Bitcoin libraries per programming language. This paper evaluates the API usability of Libbitcoin (https://github.com/Libbitcoin accessed 03.03.2020), an actively maintained open-source Bitcoin implementation that is written in the C++ programming language. Unlike Bitcoin Core (https://github.com/Bitcoin/Bitcoin accessed 03.03.2020), Libbitcoin is intended to be used as a software library.

It is worth noting that the academic literature reports a couple of tools such as SmartCheck [34], Slither [35] and the work of Ye et al. (2020) in [36] that support the verification of smart contracts (i.e., based on Ethereum). However, it fails to show or provide software developers with static analysis tools to verify Bitcoin API's security usability. We suggest the reader go through the work of Grishchenko et al. (2018) in [37] and the work of Liu and Liu (2019) in [38] for more information about the security verification of Blockchain smart contracts, which is beyond the scope of this paper.

Table 1. Related work comparison.

Work	Aim	Related to API Usability	Related to Security APIs	Usability Heuristics and Guidelines	Detect API Misuse	Tools to Detect Misuse
[39]	programmers perception on app security responsibility	○	●	○	○	○
[40]	how developers use the static code analysis tools	○	○	○	○	●
[41]	automatic detection of API usability Problems	●	○	○	●	●
[42]	API new usage rules fixing security problems	●	●	○	●	●
[43]	Relation between developer experience and personality traits and API misuse	●	●	○	○	○
[44]	Identify API usability issues of SCrypt implementation	●	●	○	●	○
[45]	platform providing examples on the correct use of crypto APIs	●	●	○	○	○
[46]	Usability evaluation of Rust cryptographic libraries	●	●	○	●	○
[29]	Categorisation of API usability heuristics	●	○	●	●	○
[47]	Tool to help developers correctly use crypto API	○	●	○	○	●
[2]	Usability evaluation of Python cryptographic libraries	●	●	○	○	○
[48]	How to conduct developer security usability studies?	●	○	○	○	○
[49]	Security and usability impact of using immutability	●	○	○	○	○
[17]	How does API usability relate to software developers handling passwords securely?	●	●	○	○	○
[28]	Classification of security APIs	●	●	○	○	○
[50]	What obstacles do developers face when using Java crypto APIs?	●	●	○	○	◉
[51]	Is it possible to create security libraries that are easy to use?	●	●	○	○	○
[52]	A static code analysis tool to detect security vulnerabilities in PHP applications	○	○	○	○	●
[53]	A tool to detect and repair cryptographic misuse on bytecode level	◉	●	○	●	●
[27]	developer-friendly security by increasing usability	●	●	◉	○	○
[54]	Suggestion for semantic crypto API offering better usability	●	●	○	○	○
[3]	Why API usability is important, especially for security	●	●	○	○	○
[55]	Suggestion for semantic crypto API offering better usability	●	●	○	○	○
[18]	Suggestions for improving usable security for developers research	●	●	○	○	○
[56]	Detailed analysis of the current state of API usability	●	●	◉	○	○
[8]	Definition of metric for the quantitative analysis of API usability	●	○	○	◉	○
[57]	Development of a code completion tool to improve API usability	●	○	○	○	●
[58]	cryptographic libraries usability and a new tool to identify misuses	●	●	○	●	●
[16]	Design of an empirical study for assessing API usability	●	○	○	○	○
[59]	Analysis and detection of crypto API misuse in Android apps	●	●	○	●	●

●: Covered by the work; ○: Not covered by the work; ◉: Partially covered by the work.

Table 2. Individual study categories.

Class	Category	Concept	Proposals
		Usable Security Research Spectrum	
		Information Source Impact	[18–21,45,60–65]
	Learning	Gaming	[22,24,25,66]
		Carelessness	[17,22,65]
	Perception	Organisational Support	[67–71]
		Complexity	[17,43,48,65,71–76]
	Development Processes	Secure SDLC	[76–84]
		Dialectic & Comm.	[22,62,85]
		Code Auto-generation	[46,66,86]
	Libraries& APIs	Cryptography	[2,27,27,42,46,50,87–90]
		Semantic APIs	[42,54,55]
		Improving Usability	[3,18,27,44,91–93]
	Languages	Secure Programming Language	[94–101]
		Language Extensions	[102]
		Comparative Analysis	[2,103–107]
	Tools	Security Assurance	[9,10,40,41,52,63,74,108–115]
		IDE Integration	[47,66,73,86,91,109,116–124]
		Adoption Factors	[11,43,49,109,124–135]
	Implementation Choices	Code Smell	[136,137]
		Security Errors Reasons	[17,44,73,75,87,88,138–142]

*(Class column, spanning rows: **Behavioural** over Learning/Perception; **Programming** over Development Processes through Implementation Choices.)*

4. The Proposal: The Two Static Analysis Tools

For an initial evaluation of the API usability of the selected Bitcoin libraries and implementations, the process suggested by Mosqueira-Rey et al. (2018) in [29] is followed, which falls under knowability, robustness and safety. The reason to adopt Mosquera-Rey's work is the inclusiveness of the proposed heuristics and the ability to be used for Bitcoin's APIs to evaluate its security usability. Unlike, for example, the generic methodology of Wijayarathna and Arachchilage (2018) in [143] and that of Grill's et al. (2012) in [144], the latter work represents a mature stream of research in Human–Computer Interaction that dates back to the '90s (i.e., Nielsen's "heuristic evaluation" guidelines in [145]). The work of Zibran (2008) in [15] has also proposed 22 API usability factors solely on the basis of surveying the literature and "there is no indication of [the] relative significance of one factor over another" [4]. Tools such as StopMotion [41] have a limited scope of identifying general API usability problems by contrasting committed code at file-level for successive changes. It is worth emphasising that a comprehensive list of low-level API security usability heuristics are still missing.

In this research, the various Libbitcoin's APIs are compared against the heuristics defined in that work, then each API function is categorised according to one of the following classifications. Yes (●): The heuristic is fulfilled, Partially (⊙): The heuristic is partially fulfilled (e.g., some of the guidelines were followed while others are missing), and No (○): The heuristic is not fulfilled. This API comparative analysis is done through the proposed static analysis tools as well as the Cognitive Walkthrough method mentioned earlier. Reliable source code analysis, done independently from the chosen programming language, relies on a language parser that recognises the language's syntax. The parser's input is a source code file, and its output is some form of intermediate representation, typically an Abstract Syntax Tree (AST). Tools can then either analyse the AST to generate diagnostic output or change the AST and feed it to a pretty-printer as a way to automate code refactoring [146]). In this research, we used a Clang-based C++ tool. Clang provides three methods for building tools that make use of Clang's parsing functionality. First, there is LibClang, a C interface library that can be used from other tools and programming languages other than C++. A second option is Clang

Plugins, which allow integrating additional functionality into the C++ build process. Finally, there is LibTooling. LibTooling allows building standalone tools that have full access to the parsed AST. Having said that, two C++ static code analysis tools are implemented in this research using the Clang's LibTooling library for parsing and analysing the ASTs of C++ programs and could be found on Github (https://github.com/phitsc/check-cpp-api accessed 03.03.2020 https://github.com/phitsc/find-api-usage accessed 03.03.2020). The tools are:

1. *check − cpp − api*, which checks C++ applications for violations of the API heuristics and guidelines we highlighted before. The tool could be used on a single .cpp file as follows [147]:

    ```
    ./check-cpp-api -p ./build ./main.cpp
    ```

    Or, it can process all files in a directory by calling:

    ```
    find ./Libbitcoin/src/
        -name *.cpp
        -exec ./check-cpp-api
            -kc-1-1-case-type=snake
            -p ./Libbitcoin/build {} +
    ```

2. *find − api − usage*, which supports finding the best usage of those identified problematic APIs issues by *check − cpp − api*. It could be used as follows:

    ```
    ./find-api-usage
        -p Libbitcoin/build
        -fc script::is_valid
        Libbitcoin/src/chain/input.cpp
    ```

It is worth noting that while all development and testing were done on a Linux environment, no platform-specific functionality was used. Thus, building and running both tools on other platforms for which Clang is available should, therefore, be straightforward. However, the availability of a compilation database is required. To facilitate the development of these tools and help to automate the build process, a Docker (https://www.docker.com/ accessed 03.03.2020) environment, paired with some Bash and Python scripts, were developed.

check − cpp − api currently provides checks for the following Mosqueira-Rey et al. (2018) guidelines [29]:

1. *KCE − 1 − 1* Avoid cryptographic names and abbreviations. The current implementation checks for the use of abbreviations in function names by checking whether the function being processed ends with a fixed number of often-used abbreviations in a custom dictionary.
2. *KCE − 2 − 2* Use enumerations instead of booleans for options. The current implementation checks whether the function being processed has any arguments of boolean type.
3. *KC − 1 − 1* Use consistent naming. The current implementation verifies whether every function name adheres to the specified naming convention, which can be either of Hungarian (i.e., `nSomeNumber`), Camel Case (i.e., `thisIsCamelCase`), Pascal Case (i.e., `ThisIsPascalCase`) or Snake Case (i.e., `this_is_snake_case`).
4. *KC − 1 − 2* Use consistent parameter ordering. The current implementation verifies that in- and out-parameters are not intermixed, i.e., that all in-parameters come before all out-parameters or vice verse.

5. $KM - 1 - 1$ Prefer short function names. The current implementation checks that the lengths of all function names are below a specified or a predefined number of characters. The default length used is 40 characters.
6. $KM - 1 - 2$ Avoid functions with many parameters. The current implementation checks that the number of parameters of each function is below a specified or predefined value. The default value used is six parameters.
7. $KM - 1 - 3$ Avoid functions with many consecutive parameters of the same type. The current implementation, for each function, counts the number of consecutive parameters with the same type and reports occurrences where that number is larger than a specified or predefined value (i.e., threshold). The default value used is three consecutive parameters with the same type.
8. $RU - 2 - 1$ Prefer standard over exceptional processing by using optional return values. The current implementation reports all functions that have both a boolean return type and one or more out-parameters.

4.1. The Specifications and Design Objectives

Based on our research, as demonstrated in Table 3, the current static analysis tools' design objectives are:

1. Improving the alarm system by, for example, reducing false-positives or narrowing down the scope to only those actionable warnings.
2. Auto-generation of secure code. Some tools are going the extra mile to correct the existing code or to auto-generate secure code for specific tasks.
3. Some tools are focusing on a specific set of code vulnerabilities to make sure that the list of warnings is actionable. Since the scope of this research is centred around the security usability of current Bitcoin's APIs, our work (i.e., the tools we propose here) falls under this category of design objectives.
4. IDE Integration. Those tools are fully integrated into IDEs facilitating, on the spot, the identification of possible security usability problems.

Table 3. Static analysis tools: design objectives.

Objective	Proposals
Reducing No. of Alarms	[11,109,124,130–133]
Code Auto-generation	[47,66,86,91,148]
Focusing on limited vulnerabilities	[116,123,149–151]
IDE Integration	[66,73,86,91,117–122]

Our proposal (i.e., $check - cpp - api$ and $find - api - usage$), although targeting a special set of API vulnerabilities related to Bitcoin, has the vision of supporting IDE integration and auto code generation. However, those two functions have not been fully implemented yet and are considered part of this paper's future work.

An essential architectural characteristic of static code analysis tools is extensibility, where it should be straightforward to add additional checks or remove deprecated ones. This flexibility can be achieved with a generic object model that allows executing checks seamlessly. The class diagram depicted in Figure 2 presents the proposed static analysis tools' classes and their relationships to provide such extensibility. It is worth noting that both tools are based on LibTooling's `ClangTool` class. On the execution of the `ClangTool::run` method, the tools start analysing the source files passed as program arguments. Once the instantiated `MatchFinder` object finds a match, it executes the `run` method on the registered `MatchFinder::MatchCallback` object, passing the matched AST subtree as an argument. In our case, this is the `HeuristicCheckAction::run` method, which then executes the `Heuristic::check` method on all registered heuristics. The `Heuristic::check` methods,

in turn, execute all `Guideline::check` methods of the `Guideline` objects registered on their related `Heuristic` object. Finally, the `HeuristicCheckAction::run` method prints all failed checks to the console or exports them to a file in a structured file format for further processing. Since all checks may require access to certain specified command-line options, an `Options` object is passed through the whole call-chain.
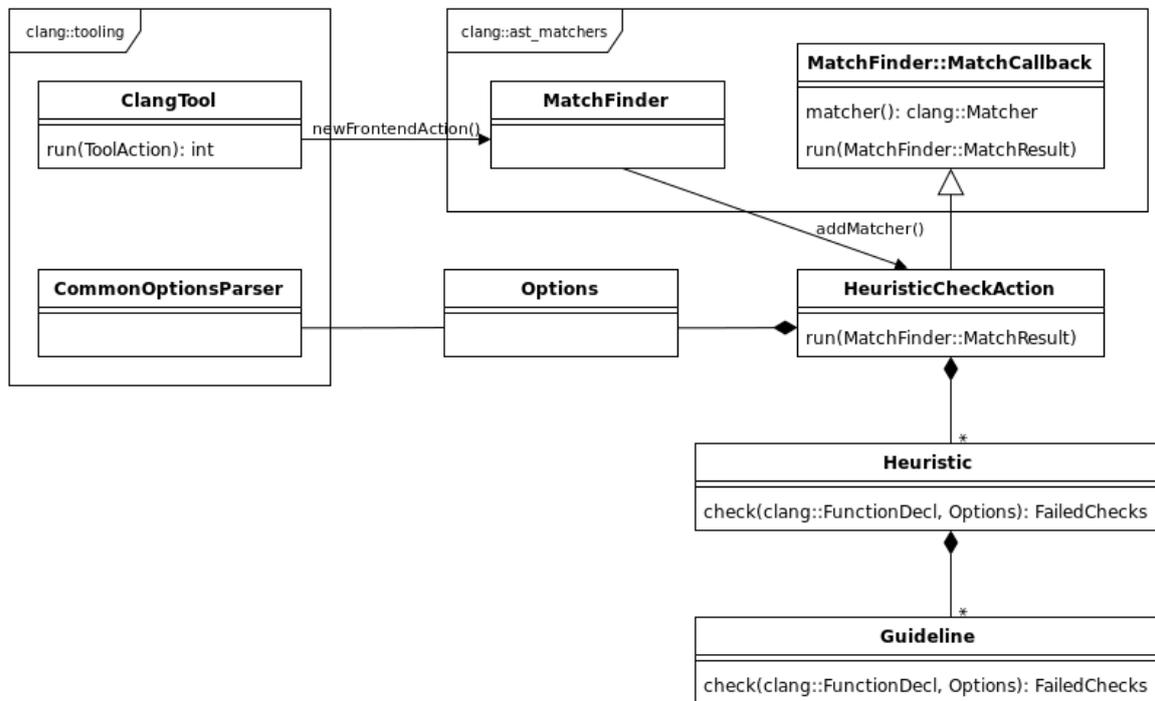

Figure 2. *check − cpp − api*: The building blocks.

5. Evaluation of the Results

In this subsection, we evaluate two overly important points; firstly, as Table 4 shows, how Libbitcoin complies with the API usability heuristics defined before by Mosqueira-Rey et al. (2018). This is simply accomplished by investigating the source code of Libbitcoin, using a hybrid of automated and Cognitive Walkthrough methodologies. A detailed assessment of Libbitcoin APIs' usability follows in the following subsections. These assessments are based on the source code, in-code API documentation (if any) and any additional official API documentation. Only those API methods and functions with public accessibility were considered for this evaluation. The tools used for the evaluation include the standard Linux tools such as *find* and *grep*, the code searching tool *ag* (i.e., The Silver Searcher (https://github.com/ggreer/the_silver_searcher accessed 03.03.2020)) and the *check − cpp − api* tool developed in this paper. Secondly, we assess the API utilisation in existing open-source projects to investigate how software developers utilise those APIs in their code.

Table 4. Libbitcoin API usability.

	Guidelines
Covered by this work	KCE-1, KCS-2, KCF-2, KC-1, KC-2, KM-1, KM-2, RU-2, SUC-1
Partially covered by this work	KCE-2, KHS-1, KHS-3, KHS-5, RU-1, RU-3
Not covered by this work	KHS-4, SUA-1

5.1. Libbitcoin APIs: Evaluation

The Libbitcoin libraries are available on GitHub (https://github.com/Libbitcoin/ accessed 03.03.2020). The developer's documentation can be found on Libbitcoin's Wiki page. This evaluation is based on those Libbitcoin Git repositories. Running Tokei (https://github.com/Aaronepower/tokei accessed 03.03.2020) on those repositories results in a total of 1393 C++ source and header files and 169689 physical lines of C++ source code. The 88679 of these physical C++ lines of source code belong to the foundation Libbitcoin library, which was renamed to Libbitcoin-system shortly after this evaluation had been concluded. It is worth mentioning that some of the below evaluation uses the Cognitive Walkthrough methodology of Callaghan (2010) in [30], where the authors of the paper "walk through" the Libbitcoin's code line by line to gain an understanding of the library concerning the heuristic under investigation.

KCE-1 Names should be self-explanatory

Using the Cognitive Walkthrough methodology shows that Libbitcoin's API uses expressive function names that, mostly, avoid the use of abbreviations. While the use of abbreviations such as `multisig` (i.e., Multisignature) is domain terminology and, therefore, is sensible to use, the use of `min_version` and `max_version` in *message/alert_payload.hpp* or `max_money` in *Bitcoin/settings.hpp* is slightly inconsistent compared to the 'minimum' and 'maximum' used in other parts of the API.

KCE-2 Data types should be as specific as possible to make the code more readable

The choice of data types is exemplary in Libbitcoin. The object model is well defined, with dedicated types for each entity. The numeric data types used have a predefined bit width. The standard C++ data types and classes are used appropriately. However, the boolean flags are used plentifully throughout the API (see Listing 1). The meanings of the boolean flags are difficult to remember, making it necessary to regularly look up their purpose in the documentation. This is a usability problem.

Listing 1. *check-cpp-api* output for KCE-2 heuristic (truncated).

```
> find Libbitcoin -all -name '*.hpp'
  -exec check -cpp -api
  -kc -1 -1 -case -type = snake {} +
  | grep KCE -2
.../Bitcoin/chain/block.hpp:93:
  KCE -2 -2: boolean parameter
.../Bitcoin/chain/header.hpp:109:
  KCE -2 -2: boolean parameter
.../Bitcoin/chain/input.hpp:90:
  KCE -2 -2: boolean parameter
...
```

To investigate this usability problem further, Listing 2 shows one of the Libbitcoin's functions that uses a boolean flag. As one can see in Listing 3, the same function could be made more expressive using an enumeration. Finally, Listing 4 contrasts the usage of these two functions illustrating the usability advantage of using enumerations for flags, compared to boolean values, as suggested by our tools.

Listing 2. Boolean flag used in `parse_signature` in *math/elliptic_curve.hpp*.

```
BC_API bool parse_signature(
  ec_signature& out,
  const der_signature& der_signature,
  bool strict);
```

Listing 3. Hypothetical `parse_signature` using an enumeration.

```
enum class der_enforcement {
  lax ,
  strict
};

BC_API bool parse_signature (
  ec_signature& out ,
  const der_signature& der_signature ,
  der_enforcement der_enforcement );
```

Listing 4. Using boolean vs. enumeration for flags.

```
// true: lax or strict?
parse_signature(ec_sig, der_sig, true);

parse_signature(ec_sig, der_sig,
  der_enforcement::strict);
```

KCS-2 When reading code that uses the API, it should be easy to understand what that code does

We observe that although Libbitcoin does not support the chaining of methods, this does not adversely affect the readability of Libbitcoin's code. Libbitcoin code is easy to read because its types, methods and functions are meaningfully and consistently named. In Libbitcoin's codebase, there is a slight majority of positive over negative conditionals. Furthermore, Libbitcoin seems to adhere to C++ standard conventions as much as possible, which makes its code easy to read for anyone who is familiar with modern C++ code.

KCF-2 Functions should perform only the tasks described in their names

One of the guidelines in KCF-2 suggests that methods should not have side effects. One instance of side effects in C++ is methods that change the parameterised object(s) [152]. C++ offers the `const` qualifier to tell both the compiler and the users of an API that particular methods are free of side effects (i.e., that their execution will not change the observable state of their related objects). Libbitcoin uses `const` meticulously in method declarations. All property accessors and predicate methods (i.e., methods starting with `is_` and `has_`) are `const`-qualified. The same is true for other methods whose names indicate that they should not influence an object's state.

KC-1 The API should be consistent with itself

Libbitcoin's naming is mostly consistent. The snake case naming convention is consistently used. Parameters seem to be consistently ordered. Out-parameters typically appear before in-parameters in function declarations throughout Libbitcoin. In the rare instances where this is not the case, such as with some of the `network_address` methods in *message/network_address.hpp*, the deviation is consistent. However, a few API inconsistencies do exist. Errors, for example, are not handled consistently throughout the whole codebase. Some functions use a boolean return value to indicate success/failure of a function, as shown in Listing 5, while others return a failure code, as seen in Listing 6, and yet others might throw an exception, as seen in Listing 7.

Listing 5. `decode_base58` in *formats/base_58.hpp*.

```
BC_API bool decode_base58 (
  data_chunk& out ,
  const std::string& in );
```

Listing 6. Some of block's methods return an error code in *chain/block.hpp*.

```
code accept_transactions(
  const chain_state& state) const;
code connect() const;
code connect(
  const chain_state& state) const;
code connect_transactions(
  const chain_state& state) const;
```

Listing 7. `to_utf8` throws an exception in *unicode/unicode.cpp*.

```
size_t to_utf8(char out[], size_t out_bytes,
  const wchar_t in[], size_t in_chars) {
  ...
  if (bytes > out_bytes)
    throw std::ios_base::failure(
      ''utf8 buffer is too small'');

  return bytes;
}
```

Additionally, some objects can be in an invalid state. While most of these objects provide an `is_valid` method (e.g., `chain::header` in *chain/header.hpp*), others overload `operator bool` (e.g., `wallet::stealth_address` in *wallet/stealth_address.hpp*).

KC-2 The API should be consistent with standard conventions

The standard C++ language does not prescribe any standard for formatting code or naming entities. However, some naming and formatting standards have been established. In any case, a good recommendation is to be consistent within a codebase with a specific standard that has been chosen [153]. Libbitcoin mostly adheres to this recommendation. Property accessors mostly use the `property/set_property` naming convention. In-parameters are passed by value for simple types, constant reference for complex types, or rvalue-reference for consumed parameters. The out-parameters are passed by reference. Non-static property accessors for querying property values (i.e., getters) and predicates (i.e., `is_`) are properly qualified with the `const` keyword in most cases. While a few minor inconsistencies do exist, they look like oversights rather than deliberate deviations. The `BC_PROPERTY_GET_REF` macro defined in *config/printer.hpp* results in property getters with a `get_` prefix (instead of no prefix) and without the `const` qualifier. The `BC_PROPERTY` macro defined in *config/parameter.hpp* also results in property getters with a `get_` prefix. However, this macro does apply the `const` qualifier to the resulting property accessors. Further examples are the use of pass by constant value in the `header_organizer` constructor in *organizers/header_organizer.hpp* or the pass by value instead of pass by constant reference of `data_slice` in most of the APIs.

KM-1 The API should be easy to remember

Mosqueira-Rey et al. (2018) in [29] suggest that methods should not have long names. However, neither Mosqueira-Rey et al. (2018) nor the sources they reference for that guideline provide a concrete value of a sensible maximum length for method names. The work of Scheller and Kühn in [153] on page 15, which Mosqueira-Rey et al. (2018) reference for that guideline, suggests that "the length of names does not have a significant impact on usability". The example name that was given by Mosqueira-Rey et al. (2018) for that guideline seems difficult, long and does not make much sense. The longest methods in Libbitcoin's public API are `is_pay_witness_script_hash_pattern` in *chain/script.hpp*,

`transaction_pool_fetch_transaction`, `transaction_pool_fetch_transaction2`, `blockchain_fetch_transaction_index`, and `block-chain_fetch_unspent_outputs` in *client/obelisk_client.hpp*. The names of those functions are expressive and consistent with the names of other, similar functions, and should, therefore, be easy to remember despite their length.

It should be easier to remember how to use functions with fewer parameters compared to ones that have many parameters. The suggestion is to aim for four parameters or fewer, which seems a reasonable number to aim for but not a real hard limit [29]. This study assumes at most six parameters to be a good compromise between usability and practicability. The method with the most parameters in Libbitcoin is the `version` constructor in *message/version.hpp*, which requires nine arguments. There are two functions in Libbitcoin's public API. with eight parameters each, namely `check_signature` and `create_endorsement` in *chain/script.hpp*, both of which have eight parameters but require only six (i.e., the last two are optional parameters with default arguments). Three functions take seven arguments, including `get_map` in *chain/chain_state.hpp*, the `program` constructor in *machine/program.hpp*, `create_key_pair` in *wallet/encrypted_keys.hpp*. The `program` constructor accepts one optional argument and `create_key_pair` is commented to be deprecated.

Confusing arguments of the same data type can be particularly unsafe. Normally, compilers cannot detect such mistakes. Preconditions could be used as a countermeasure provided that the values of the concerned parameters do not overlap. What is more likely, however, is that arguments that are passed in the wrong order manifest as run-time errors. In the worst case, mixed up arguments having the same data type exhibit no errors. Instead, they make the application behave in a correct but undesired way. This study, therefore, investigates whether Libbitcoin had any APIs accepting more than three consecutive arguments with the same data type. Libbitcoin has only a single such function in its public API, namely the `block_database` constructor in *databases/block_database.hpp*, which has four consecutive parameters of type path, followed by two parameters of type `size_t`. Listing 8 shows the output of check-cpp-api that was used for the above analysis.

Listing 8. *check-cpp-api* output for the KM-1 heuristic (truncated).

```
> find Libbitcoin - all - name '*. hpp '
  - exec check - cpp - api
  - kc -1 -1 - case - type = snake {} +
  | grep KM -1
.../ Bitcoin / chain / chain_state . hpp :130:
  KM -1 -2: too many parameters
.../ Bitcoin / chain / chain_state . hpp :223:
  KM -1 -3: too many cons . params of same type
.../ Bitcoin / chain / script . hpp :136: KM -1 -2:
  too many parameters
.../ Bitcoin / chain / script . hpp :176: KM -1 -1:
  long function name
...
```

KM-2 The API should follow the terminology of the field

Although Libbitcoin mostly adheres to domain terminology, a few deviations do exist. The `header` class in *chain/header.hpp* exhibits the `Merkle` property. However, the related domain terminology is Merkle Root, as can be seen in Figure 1. Naming the property `merkle_root` would, therefore, be more in line with Bitcoin terminology. Indeed, the `block` class in *chain/block.hpp* does exhibit the two-member functions `generate_merkle_root` and `is_valid_merkle_root`, which highlights the inconsistency.

KHS-1 Every element of the API should be documented

The Libbitcoin's source code does contain some documentation. However, that documentation cannot be considered usable API reference documentation. While some documenting comments describe the purpose of the API method, most such comments are merely brief. Furthermore, some documenting comments seem contradictory, such as the one in Listing 9. If the terminology "is set" is appropriate, then why not call the function `is_set` or `is_fork_set`?

Listing 9. `chain_state::is_enabled` in *chain/chain_state.hpp*.

```
// Check if the block's fork is set
bool is_enabled(
    machine::rule_fork fork) const;
```

Although most of Libbitcoin's source code does not use structured comment blocks usable for generating reference documentation, some of it does. It is, however, neither complete nor up-to-date. An example is shown in Listing 10. Only one of the function's three parameters is documented. Furthermore, the parameter name in the documentation does not match the function's actual parameter name. Additionally, although the `signature` is an out-parameter, it is marked 'in' in the structured documentation comment.

Listing 10. `sign_message` in *wallet/message.hpp*.

```
/**
 * Signs a message using deterministic
 * signature.
 * @param[in] out_signature The
 * elements of in Bitcoin's own
 * format. This should be base64
 * encoded for presentation to the user.
 * @return true if wif is valid and
 * signature encoding is successful.
 */
BC_API bool sign_message(
  message_signature& signature,
  data_slice message,
  const ec_private& secret);
```

Some API documentation has misleading parameter annotations. The `@param[in]` annotation in Listing 11 implies that the `list` parameter is an input parameter into the function (http://www.doxygen.nl/manual/commands.html accessed 03.03.2020). However, the fact that it is passed by reference, instead of by constant reference, indicates that the function might serve as both an input and an output parameter.

Listing 11. `distinct` in *utility/collection.hpp*.

```
/**
 * Obtain the sorted distinct
 * elements of the list.
 * @param <Element> The list element type.
 * @param[in] list The list.
 * @return The sorted list reduced to
 * its distinct elements.
 */
template <typename Element>
```

```
std::vector<Element>& distinct(
    std::vector<Element>& list);
```

Similarly, the `key` parameter of `find_pair_position` function declared in *collection.hpp* is annotated as an input parameter, as seen in Listing 12, but passed by reference in the function declaration. However, the function's implementation in *collection.ipp* accepts the `key` parameter by constant reference, as Listing 13 shows. The reason why this works is that the function in Listing 13, whose definition is included in *collection.hpp* using an `#include` statement, overloads the function with the same name in Listing 12.

Listing 12. `find_pair_position` declaration in *utility/collection.hpp*.

```
/**
 * Find the position of a pair in an
 * ordered list.
 * @param   <Pair> The type of list
 * member elements
 * @param[in] list The list to search.
 * @param[in] key The key to the element
 * to find.
 * @return The position or -1 if not found.
 */
template <typename Pair, typename Key>
int find_pair_position(
  const std::vector<const Pair>& list,
  Key& key);
```

Listing 13. `find_pair_position` definition in *utility/collection.hpp*.

```
template <typename Pair, typename Key>
int find_pair_position(
  const std::vector<Pair>& list,
  const Key& key) {
    const auto predicate =
      [&] (const Pair& pair) {
        return pair.first == key;
      };
    auto it = std::find_if(
      list.begin(), list.end(), predicate);
    if (it == list.end())
      return -1;
    // Unsafe for use with lists greater
    // than max_int32 in size.
    Bitcoin_ASSERT(list.size() <= max_int32);
    return static_cast<int>(
      distance(list.begin(), it));
}
```

We conclude that without meaningful documentation that makes it easy on developers to use the APIs, it is hard to maintain the security of the code. It is worth noting that the learning resources of security have been studied in recent years by the academic community of usable security.

KHS-3 The API should properly identify deprecated classes and methods

　　Libbitcoin does identify some API functions in *utility/pseudo_random.hpp* and some methods of the `transaction` class in *chain/transaction.hpp* as deprecated. Although Libbitcoin defines the `BC_DEPRECATED` macro in *Bitcoin/define.hpp* that would produce a compiler warning when a deprecated function is used, that macro is not used anywhere.

KHS-4 The API should supply helpful error information and, if possible, suggest a solution

　　Errors can be handled using various strategies in C++. Returning an error code and throwing exceptions are two such strategies. The Libbitcoin Wiki provides no treatment of how errors are handled within the library or how to deal with them when using the library. However, inspection of Libbitcoin's APIs and implementation shows that Libbitcoin handles errors in three different ways. Some functions throw exceptions, some return a boolean value to indicate success or error and some return an error code. An advantage of exceptions over boolean or integer return values is that exception types already provide some information about the causes of errors/exceptions. Exceptions also make it more difficult to ignore erroneous events. However, Libbitcoin makes only little use of exceptions. Instead, it primarily relies on the returning of error codes and boolean error indications.

KHS-5 The API documentation should include code samples for the most common scenarios

　　Although some of Libbitcoin's libraries contain an example directory, these directories contain only a few short examples. Libbitcoin's Wiki provides 14 web pages with developer documentation on various topics. Additionally, 11 web pages with code examples related to the developer documentation topics are available on the Wiki. However, some of these examples use API functions that have been marked deprecated. One such example is the use of `pseudo_random_fill` in the *SerializedData* example.

RU-1 The API should allow detecting and managing errors without breaking the execution or leaving the error undetected

　　Until C++ gets proper support for contract programming [154], the verification of pre-conditions, post-conditions and invariants are sometimes checked using the assert macro. Software developers also often use assertions to verify assumptions [155]. Libbitcoin seems to be using assertions for all these cases in various parts of the library. However, it does not do that consistently. Furthermore, in some places, assertions seem to be used inappropriately. Because assertions are not in effect in production builds, assertions should not be used to test conditions that may just as well happen in production. Listing 14 shows an assert that halts a non-production version of an application using Libbitcoin in case transaction store corruption is detected. However, the assumption is that the same condition might occur in production builds, which would lead to inconsistent results being returned.

Listing 14. Testing store corruption but only in non-production builds in *interface/block_chain.cpp*.

```
block_const_ptr block_chain::get_block(
  size_t height, bool witness,
  bool candidate) const
{
  ...
  const auto result =
    database_.blocks().get(
      height, candidate);
  ...
  transaction::list txs;
  ...
```

```
  // False implies store corruption.
  DEBUG_ONLY(const auto value =)
    get_transactions(
      txs, result, witness);
  Bitcoin_ASSERT(value);

  // Use non-const header copy to
  // obtain move construction for txs.

  auto header = result.header();
  return std::make_shared<const block>
  (std::move(header),std::move(txs));
}
```

A questionable use of `assert` is to check pointer variables before they are dereferenced, as can be found in Libbitcoin in a small number of cases. Listing 15 shows one such example.

Listing 15. Dereferencing a pointer variable that might be a `nullptr` in *pools/header_entry.cpp*.

```
// Not callable if the entry is
// a search key
const hash_digest& header_entry::
                   parent() const
{
    Bitcoin_ASSERT(header_);
    return header_->previous_block_hash();
}
```

Because Libbitcoin libraries are programmed in C++, the use of most data types will be checked at compilation time rather than at run-time. Libbitcoin also uses `enum class` in many but not all cases. `enum class` was introduced in C++ 11 (http://www.stroustrup.com/C++11FAQ.html#enum accessed 03.03.2020) to improve the type safety of enumerated types.

RU-2 The API should facilitate managing non-common but correct situations without generating exceptions or forcing users to catch them

Although Listing 16 might suggest that there is potential for using optional return values in Libbitcoin, most of them would not be particularly beneficial. There are only a few cases in the Libbitcoin APIs where an optional value could be used instead of using the combination of an out-parameter and a boolean return value indicating success or failure. However, all of them are used to handle exceptional situations where continuing on the standard program path makes no sense. While the use of `std::optional` may offer some slight advantage over a combination of the boolean return value and out-parameter, such as, for example, the possibility to make the returned value constant, it does not simplify the client code significantly.

Listing 16. *check-cpp-api* output for the RU-2 heuristic (truncated).

```
> find Libbitcoin-all -name '*.hpp'
  -exec check-cpp-api
  -kc-1-1-case-type=snake {} +
  | grep RU-2
.../Bitcoin/chain/block.hpp:98:
  RU-2: omission to use optional?
.../Bitcoin/chain/block.hpp:99:
  RU-2: omission to use optional?
```

```
.../ Bitcoin / chain / compact . hpp :51:
  RU -2: omission to use optional?
.../ Bitcoin / chain / header . hpp :113:
  RU -2: omission to use optional?
...
```

RU-3 The API should not expose vulnerabilities that would allow users to make errors

While Libbitcoin seems to handle element accessibility purposefully, there are some class members whose accessibility could be further restricted. Various classes have protected methods, e.g., `operation` in *machine/operation.hpp* or `chain_state` in *chain/chain_state.hpp*, although they do not seem to be intended to serve as base classes. Sometimes compiler bugs mandate certain members to be protected although they could be private from a language point of view. However, such cases should visibly be documented in the code. Another case where accessibility should be further restricted is the `BC_PROPERTY_GET_REF` macro shown in Listing 17. The macro generates property accessors that return non-const references to private member variables, defeating the purpose of making these member variables private in the first place. The `BC_PROPERTY_GET_REF` macro, however, is only used in a single non-critical class of Libbitcoin.

Listing 17. `BC_PROPERTY_GET_REF` exposing private member by non-const reference in *config/printer.hpp*.

```
#define BC_PROPERTY_GET_REF(type, name)
  public: virtual type& get_##name() {
    return name##_;
  }
  private: type name##_
```

Immutability is addressed nicely in Libbitcoin's API. For example, those methods that do not need to change their related object, such as property getters, are properly qualified with the `const` keyword. However, immutability could be further promoted in client code by using optional return values instead of out-parameters and boolean return values. With a boolean return value and an out-parameter, a mutable temporary variable must be created and passed into the function as Listing 18 shows. Listing 19 demonstrates how the temporary variable could be made immutable with the help of an optional return value. By doing this, an added benefit would be allowing the use of type inference which further enhances the usability of the API.

Listing 18. `create_ephemeral_key` returning boolean success or failure value in *math/stealth.hpp*.

```
BC_API bool create_ephemeral_key (
  ec_secret& out_secret ,
  const data_chunk& seed );
...
ec_secret ephemeral_private ;
if ( create_ephemeral_key (
      ephemeral_private , seed ))
  initialize ( ephemeral_private ,
    address , seed , filter );
```

Listing 19. Hypothetical `create_ephemeral_key` alternative returning an optional value in *math/stealth.hpp*.

```
BC_API std::optional<ec_secret>
  create_ephemeral_key(
    const data_chunk& seed);
...
if (const auto ephemeral_private =
      create_ephemeral_key(seed))
  initialize(*ephemeral_private,
    address, seed, filter);
```

There are no obvious threading issues in Libbitcoin from a API usability point of view. There are also no APIs in Libbitcoin returning allocated memory that must be manually freed by a user of the API. Furthermore, neither *cppcheck* (http://cppcheck.sourceforge.net/ accessed 03.03.2020) nor Clang Static Analyzer (https://clang-analyzer.llvm.org/ accessed 03.03.2020) reported any memory-related issues.

SUC-1 The API should not compromise the confidentiality of the users' personal information

Libbitcoin does not seem to acquire, store or process any personal information beyond what is necessary to fulfil its intended purpose. The sole information that could be considered personal is the one related to the wallet functionality which makes sense to use. All other data is related to the public blockchain.

SUA-1 The API should not compromise the security of the users' assets

Not compromising the security of user assets should be a primary goal of any Bitcoin library. However, assessing whether an API constitutes a risk to user assets seems challenging when evaluating the API alone, not along with its implementation. For security-critical algorithms, including the ones related to AES, ECDSA, RIPEMD, SHA and scrypt, Libbitcoin uses external implementations. Libbitcoin provides unit tests for a lot of its code. However, the test coverage should be increased to include more error and corner cases.

5.2. Open-Source Projects: Libbitcoin APIs Evaluation

It is worth emphasising that the versification of how secure Libbitcoin's APIs are is outside the scope of this research. Having evaluated Libbitcoin's APIs in the previous sections, it is now time to investigate how such APIs are used in production. This subsection demonstrates the results of investigating the utilisation of Libbitcoin in the following open-source projects concerning the findings described in the previous subsections:

1. https://github.com/Libbitcoin/Libbitcoin-explorer.git with Commit ID of f4dd566fbce806f3e622
2. https://github.com/Libbitcoin/Libbitcoin-node.git with Commit ID of 791f5ab5ab5eb5f01b09
3. https://github.com/Libbitcoin/Libbitcoin-server.git with Commit ID of 3118b3b4495cc8a94894
4. https://github.com/mvs-org/metaverse.git with Commit ID of 8dec1d81fe243f6a27e1
5. https://github.com/joinparty/joinparty.git with Commit ID of 258c7419dec7ee193452.

KCE-2: Boolean Flags

As pointed out before, the boolean parameters appear plentifully in the Libbitcoin library APIs. The reason why boolean flags in APIs are problematic has been already discussed. Table 5 lists the Libbitcoin functions that violate guideline KCE-2-2 and are used in one of the evaluated projects. This subsection describes how the evaluated projects pass boolean arguments to those functions. While using an enumeration, as described before, would often be preferable, using a meaningful named variable is the best case in the given situation. The worse, but tolerable, case is a comment at the call site indicating the meaning of the parameter. The worst cases are situations where the code

surrounding the call site needs to be analysed to understand why true or false was passed to a function. The ultimate worst case is where a boolean literal is passed and software developers working on the code must look up the API documentation or an APIs implementation to understand what a boolean flag is used for.

`authenticator::apply` is used in `heartbeat_service::bind`, `query_service::bind`, `block_service::bind` and `transaction_service::bind`, both in Libbitcoin server (see Listing 20) and Metaverse. Each of these services is instantiated twice in `server_node`, once for each value of the boolean flag. While the flag's name might make the flag's meaning obvious enough, its apparent security-relevance suggests making it more obvious with the help of an enumeration. However, the examined usages were non-critical.

`block_chain::get_top` is used in `full_node::handle_running` in the Libbitcoin node (see Listing 21). In the evaluated code, the flag's meaning is made obvious through the naming of the additional parameter passed to `block_chain::get_top`. `block_chain::fetch_block` is used in `blockchain::fetch_block_by_hash` and `blockchain::fetch_block_by_height` in Libbitcoin server, as well as in `protocol_block_out::send_next_data` of Libbitcoin node. The `fetch_block` method of the `safe_chain` class, which `block_chain` derives from, is used both in the Libbitcoin server and the Libbitcoin node. In the Libbitcoin server, the flag's meaning is made obvious with the help of an appropriately named local variable whose value is calculated right before `block_chain::fetch_block` is executed. In the Libbitcoin node, boolean literals are passed whose purpose can merely be derived from the case label related to the blocks the methods are called in.

Table 5. Usage of methods with boolean flags.

API	Libbitcoin Explorer	Libbitcoin Node	Libbitcoin Server	Metaverse	Joinparty
`authenticator::apply`	○	○	●	●	○
`block_chain::fetch_block`	○	○	●	○	○
`block_chain::fetch_transaction`	○	○	●	○	○
`block_chain::get_top`	○	●	○	○	○
`create_key_pair`	●	○	○	○	○
`decode_base10`	●	○	○	●	○
`deserialize`	●	○	○	●	○
`ec_private::ec_private`	●	○	○	○	○
`ec_public::ec_public`	●	○	○	●	○
`initialize`	●	●	●	○	○
`parse_signature`	●	○	○	●	●
`payment_record::to_data`	○	○	●	○	○
`property_tree`	●	○	○	○	○
`script::from_data`	○	○	○	●	●
`split`	●	○	○	●	○
`transaction::serialized_size`	○	○	○	○	●
`wallet::sign_message`	○	○	○	○	●

●: Used; ○: Not Used.

Listing 20. Usages of `authenticator::apply` in the Libbitcoin server (truncated).

```
> find Libbitcoin-server -name '*.cpp'
  -exec find-api-usage
  -function-call='authenticator::apply'
  {} +
```

```
.../src/services/query_service.cpp:88:
  this->authenticator_.apply(router,
    domain, this->secure_)
.../src/services/block_service.cpp:96:
  this->authenticator_.apply(xpub,
    domain, this->secure_)
...
```

Listing 21. Usages of `block_chain::get_top` in Libbitcoin node.

```
> find Libbitcoin-node -name '*.cpp'
  -exec find-api-usage
  -function-call='block_chain::get_top'
  {} +
.../src/full_node.cpp:117:
  this->chain_.get_top(top_confirmed,
    false)
.../src/full_node.cpp:130:
  this->chain_.get_top(top_candidate,
    true)
```

`block_chain::fetch_transaction` requires two flags to be passed and is used in `blockchain::fetch_transaction`, `blockchain::fetch_transaction1`, `transaction_pool::fetch_transaction` and `transaction_pool::fetch_transaction2` in Libbitcoin server. The meaning of the flags is merely described in the examined code through some comments above the method invocation (see Listing 22). Without consulting the method's declaration, it is not obvious that the comment above the method invocation is related to these two boolean flags. Furthermore, comments always pose the risk of diverging with the code they describe. In the Libbitcoin node, `block_chain::fetch_transaction` is used in `protocol_transaction_out::send_next_data` with boolean literals. Again, the purpose of the boolean flags passed can merely be derived from the case label related to the blocks the methods are called in.

Listing 22. Use of boolean flags in `block_chain::fetch_transaction`.

```
/*
 * The response is restricted to the
 * confirmed transactions.
 *
 * This response excludes witness data
 * so as not to break old the parsers.
 */
node.chain().fetch_transaction(
  hash, true, false,
  std::bind(
    &blockchain::transaction_fetched,
    _1, _2, _3, _4, request, handler));
```

`parse_signature` is used in `input_validate::invoke` in Libbitcoin explorer, in `sort_multi_sigs` and `signmultisigtx::invoke` in Metaverse and in `get_ec_signature` in Joinparty. In `input_validate::invoke`, the flag is passed using a local variable that is initialised with a boolean literal and whose name gives a hint about its use. In both `sort_multi_sigs` and `signmultisigtx::invoke`, local variables are used that are initialised right before the function

call. In `get_ec_signature`, a boolean literal is passed without further comment about the argument's purpose.

In both the Libbitcoin explorer and Metaverse, the boolean flag to `decode_base10` is not passed explicitly but its default value is used. In the Libbitcoin explorer, `ec_public::ec_public` is used in `ec_add::invoke`, `ec_decompress::invoke`, `ec_multiply::invoke`, `ec_to_public::invoke` and `ek_public_to_ec::invoke`. In three cases, locally initialised and appropriately named variables are used to pass the flag. In `ec_multiply::invoke`, the flag is passed by calling a function that indicates the flag's purpose. In `ec_decompress::invoke`, a boolean literal is passed and the class name `ec_decompress` is the only indication of what the flag's purpose might be. In Metaverse, `ec_public::ec_public` is used in `getnewaddress::invoke`, which just passes a boolean literal. While there is a code comment right before the constructor call, it is not obvious that it relates to the boolean flag.

`wallet::sign_message` is used in `get_encoded_signed_message` in Joinparty. A boolean literal is passed to `wallet::sign_message` without any indication about the argument's purpose. `create_key_pair` is used in `ek_address::invoke`, `ek_new::invoke` and `commands::ek_public::invoke` in Libbitcoin explorer (see Listing 23). In all three cases, a locally initialised and appropriately named variable is used to pass the flag.

Listing 23. Usages of `create_key_pair` in Libbitcoin explorer (truncated).

```
> find Libbitcoin-explorer -name '*.cpp'
  -exec find-api-usage
  -function-call='create_key_pair'
  {} +
.../src/commands/ek-new.cpp:49:
  create_key_pair(secret, unused, ...)
.../src/commands/ek-public.cpp:52:
  create_key_pair(unused1, key, ...)
.../src/commands/ek-address.cpp:50:
  create_key_pair(unused, point, ...)
```

`ec_private::ec_private` is used in `ec_to_wif::invoke` in Libbitcoin explorer. A locally initialised and appropriately named variable is used to pass the flag.

Both the Libbitcoin explorer and Metaverse use `deserialize` in `operator>>` related to the `byte` class. In both cases, the flag is passed as a boolean literal without any further hint of what the flag's meaning might be. In the Libbitcoin explorer, `split` is only used without its flag parameter, which is initialised from its default value. Metaverse, as well, uses `split` with the flag's default value into many places. However, in Metaverse, there are also multiple cases where boolean literals are explicitly passed to `split`. While the flag's purpose is further commented in one case, in all other cases its meaning is not apparent.

`property_tree` is used in `fetch_block::invoke`, `fetch_tx::invoke`, `fetch_utxo::invoke` and `tx_decode::invoke` in Libbitcoin explorer. In all cases, a local variable is used to pass the flag, which is initialised close to the function invocation. In each case, an enumeration is used to initialise this local variable, albeit one with three possible values.

`transaction::serialized_size` is called in `Wallet::create_and_broadcast_transaction` and `Wallet::create_coin_join_transaction` in Joinparty. In both cases, no arguments are explicitly passed to `transaction::serialized_size` and the method's default arguments are used instead. In all places where `script::from_data` is used in Metaverse and Joinparty, its flag option is passed as a boolean literal without any further hints about its purpose.

`payment_record::to_data` is used in `blockchain::history_fetched` and `blockchain::stealth_fetched` in Libbitcoin server. In both cases, the flag is passed as a boolean literal without any further hints about its purpose.

initialize, a free function related to Libbitcoin's logging functionality. While its flag option is passed as a named local variable in both Libbitcoin node and Libbitcoin server, it is passed as a boolean literal without any further comment about its function in Libbitcoin explorer.

KC-1: Unhandled Error Results

In contrast to exceptions, boolean error results and error result codes are easier to ignore, be it on purpose or by accident. This study, therefore, examined the usage of functions and methods flagging success or failure by returning a boolean value or error code of type `std::error_code` (used as code in Libbitcoin with the help of a typedef). Table 6 lists some of those functions (The full set of functions is available upon request should the corresponding author is contacted) and methods of Libbitcoin's public API that fall into this category and are used in one of the evaluated projects.

Table 6. Functions with a boolean error result.

API	Libbitcoin Explorer	Libbitcoin Node	Libbitcoin Server	Metaverse	Joinparty
`Bitcoin_uri::set_address(const std::string&)`	●	○	○	●	○
`create_key_pair`	●	○	○	○	○
`create_stealth_data`	○	○	○	●	○
`create_token`	●	○	○	○	○
`decode_base10`	●	○	○	●	○
`decode_base16`	○	○	○	●	●
`decode_base58`	●	○	○	●	○
`decode_base64`	●	○	○	●	●
`decrypt`	●	○	○	○	○
`ec_add`	●	○	○	●	●
`ec_multiply`	●	○	○	●	●
`encode_signature`	○	○	○	○	●
`encrypt`	●	○	○	○	○
`extract_ephemeral_key`	○	○	○	●	○
`parse_signature`	○	○	○	●	●
`parser::get_option`	●	●	●	●	○
`png::write_png`	●	○	○	○	○
`point::from_data`	○	○	●	●	○
`property_tree`	●	○	○	○	○
`script::check_signature`	●	○	○	●	○
`script::create_endorsement`	●	○	○	●	●
`script::from_data`	○	○	○	○	●
`script::verify`	○	○	○	○	●
`secret_to_public`	●	○	○	●	●
`sign_message`	●	○	○	○	●
`sign`	○	○	○	○	●
`to_stealth_prefix`	○	○	●	●	○
`uncover_stealth`	○	○	○	●	○
`verify_message`	●	○	○	○	○
`verify_signature`	○	○	○	○	●

●: Used; ○: Not Used.

In multiple cases in the Libbitcoin explorer and Metaverse, the return value for `secret_to_public` is not checked. In Joinparty, the results of `secret_to_public` calls are always ignored. `secret_to_public` internally calls `secp256k1_ec_pubkey_create` and `serialize`, both of which can

fail. Of the three overloads of `Bitcoin_uri::set_address`, only one can fail, namely, the one which is called with a `std::string` argument. This overload is used in the Libbitcoin explorer without checking its return value (see Listing 24).

Listing 24. Usage of `Bitcoin_uri::set_address` in Libbitcoin explorer.

```
> find Libbitcoin - explorer - name '*. cpp '
  - exec find - api - usage
  - function - call =' Bitcoin_uri :: set_address '
  {} +
.../ src / commands / uri - encode . cpp :43:
  uri . set_address ( address )
```

`create_key_pair`, `create_token`, `encrypt` and `sign_message` are used in the lib-Bitcoin explorer without their return values being checked in any of the calls. `sign_message` is also called in Joinparty, where its return value is properly evaluated.

`point::from_data` is called in one instance as a call to `output_point::from_data` (`output_point` derives from `point`) in the Libbitcoin server and Metaverse without its return value is checked. While the result of calling `decode_base16` is checked in one case in Joinparty, it is ignored in three other cases. Furthermore, none of the calls to `decode_base64` in Joinparty is checked for a failed result.

`parse_signature`, `encode_signature`, `ec_add`, and `ec_mulitply` are all used in Joinparty without their return values ever being evaluated.

In the examined client applications, the return values are checked for invocations of all other functions listed in Table 6. The invocation of any API functions and methods within Libbitcoin's libraries themselves have not been further investigated. However, doing so would be advisable, as ignored error result flags are a potential source of security-critical bugs.

KC-2/RU-3: `BC_PROPERTY_GET_REF`

As demonstrated before, the use of the `BC_PROPERTY_GET_REF` macro results in property accessors that provide write access to private member variables. However, none of the evaluated applications makes use of the property accessors defined through the `BC_PROPERTY_GET_REF` macro. Instead, these property accessors seem to be only used by other public API methods of Libbitcoin's printer class. Consequently, the `BC_PROPERTY_GET_REF` macro should not only be changed to return a constant reference but the property accessors could even be implemented with private accessibility.

KHS-3: Deprecated Functions

Table 7 lists all of Libbitcoin's functions and methods that are labelled deprecated using a comment, and which are used by some of the examined client applications. The `BC_DEPRECATED` macro defined in *Bitcoin/define.hpp*, which should be preferred for tagging deprecated functions, is not used anywhere in Libbitcoin. Furthermore, Mosqueira-Rey et al. (2018) suggest that deprecated API elements should have accompanying documentation explaining the reasons for the deprecation and proposing viable alternatives [29]. None of the functions listed in Table 7 has such accompanying documentation.

Table 7. Deprecated functions.

API	Libbitcoin Explorer	Libbitcoin Node	Libbitcoin Server	Metaverse	Joinparty
transaction :: inputs() (non-const version)	●	○	○	○	●
transaction :: outputs() (non-const version)	●	○	○	○	●
pseudo_random()	○	○	○	●	○
ine pseudo_random(uint64_t, uint64_t)	○	○	●	○	○
pseudo_random_fill(data_chunk&)	●	○	○	●	●
pseudo_randomize(const asio:duration&, uint8_t)	○	○	○	○	○
create_key_pair(encrypted_private&,encrypted_public&, ec_compressed&,...)	●	○	○	○	○
decrypt(ec_compressed&, uint8_t&, bool&, const encrypted_public&, ...)	●	○	○	○	○

●: Used; ○: Not Used.

6. Conclusions and Future Work

This paper attempts to understand the usability issues in Bitcoin's APIs, namely in the Libbitcoin implementation and how software developers misuse the Bitcoin APIs in their code. Libbitcoin is a well-known C++ implementation of the Bitcoin system, which has been used in this project. This paper evaluates those APIs from a security usability perspective. As far as the authors of this paper know, it is the first attempt to understand how Bitcoin's APIs are used in open-source projects and how to avoid certain coding practices that could leave the application vulnerable to attacks. It is worth emphasising that while some research has been done on the usability of Bitcoin's applications from an end-user's point of view, as far as the authors of this paper know, there is no research yet that addresses usability aspects from a developer's point of view. This work proposed two static analysis tools to identify security usability concerns and suggests resolutions for such concerns. This paper comprehensively surveyed the general academic literature concerning API usability and usable security. The findings of this research has improved Libbitcoin in many places. The paper answers questions such as "How usable are Bitcoin's API libraries from a security preservative?", "How would a static analysis tool help in minimising code volubility using those identified Bitcoin API libraries?" and so forth. To be able to do that, the paper attempted to study the usability issues with identified Bitcoin's APIs, investigate the static analysis tools and how they could help to raise the awareness of Bitcoin's software developers to avoid such usable security issues and so forth.

The future direction of this paper includes fully integrating the tool in moderns IDEs, such as Eclipse, to make it easier for the developers to identify concerns on the run and get online suggestions on how to resolve those usability/security issues. A qualitative analysis of the tools needs to be carried out by surveys and questionnaires, which requires the tools to gain some popularity in the Blockchain development community before a statistically significant response is received. One of the limitations of this study is that it focuses only on Libbitcoin's C++ API implementation and the rest of available libraries are considered future work.

Author Contributions: Conceptualization, P.T. and A.A.; methodology, P.T. and A.A.; software, P.T.; validation, P.T.; formal analysis, P.T.; investigation, P.T. and A.A.; resources, P.T. and A.A.; data curation, P.T. and A.A.; writing–original draft preparation, P.T. and A.A.; writing–review and editing, A.A.; visualization, P.T. and A.A.; supervision, A.A.; project administration, A.A.; funding acquisition, P.T. and A.A. All authors have read and agreed to the published version of the manuscript.

References

1. Bevan, N. International standards for HCI and usability. *Int. J. Hum.-Comput. Stud.* **2001**, *55*, 533–552.
2. Acar, Y.; Stransky, C.; Wermke, D.; Mazurek, M.; Fahl, S. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In Proceedings of the Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017), Santa Clara, CA, USA, 12–14 July 2017; Association: Santa Clara, CA, 2017; pp. 81–95.
3. Myers, B.A.; Stylos, J. Improving API Usability. *Commun. ACM* **2016**, *59*, 62–69.
4. Zibran, M.F.; Eishita, F.Z.; Roy, C.K. Useful, But Usable? Factors Affecting the Usability of APIs. In Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE '11), Limerick, Ireland, 17–20 October 2011; IEEE Computer Society: Washington, DC, USA, 2011; pp. 151–155.
5. Acar, Y.; Fahl, S.; Mazurek, M. You Are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In Proceedings of the 2016 IEEE Cybersecurity Development (SecDev), Boston, MA, USA, 3–4 November 2016; IEEE: Boston, MA, USA, 2016; pp. 3–8.
6. Barbara, K.; Stuart, C. *Procedures for Undertaking Systematic Reviews*; EBSE Technical Report EBSE-2007-01; Keele University and University of Durham: Keele, UK, 2007.
7. Merino, L.; Ghafaria, M.; Anslow, C.; Nierstrasz, O. A Systematic Literature Review of Software Visualization Evaluation. *J. Syst. Softw.* **2018**, *144*, 165–180.
8. Rama, G.M.; Kak, A. Some Structural Measures of API Usability. *Softw. Pract. Exp.* **2015**, *45*, 75–110.
9. Jonas, W.; Volodymyr, K.; George, C.; Johannes, K. High System-Code Security with Low Overhead. In Proceedings of the 2015 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 17–21 May 2015; IEEE: San Jose, CA, USA, 2015; pp. 866–879.
10. Sadowski, C.; Aftandilian, E.; Eagle, A.; Miller-Cushon, L.; Jaspan, C. Lessons from Building Static Analysis Tools at Google. *Commun. ACM* **2018**, *61*, 58–66.
11. Muske, T.; Serebrenik, A. Survey of Approaches for Handling Static Analysis Alarms. In Proceedings of the 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM), Raleigh, NC, USA, 2–3 October 2016; pp. 157–166.
12. McLellan, S.G.; Roesler, A.W.; Tempest, J.T.; Spinuzzi, C.I. Building More Usable APIs. *IEEE Softw.* **1998**, *15*, 78–86.
13. Stylos, J.; Graf, B.; Busse, D.K.; Ziegler, C.; Ehret, R.; Karstens, J. A Case Study of API Redesign for Improved Usability. In Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, Herrsching am Ammersee, Germany, 15–19 September 2008; pp. 189–192.
14. Robillard, M.P. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Softw.* **2009**, *26*, 27–34.
15. Zibran, M.F. What Makes APIs Difficult to Use? *IJCSNS Int. J. Comput. Sci. Netw. Secur.* **2008**, *8*, 255–261.
16. Piccioni, M.; Furia, C.A.; Meyer, B. An Empirical Study of API Usability. In Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Baltimore, MD, USA, 10–11 October 2013; pp. 5–14.
17. Alena, N.; Anastasia, D.; Christian, T. Why Do Developers Get Password Storage Wrong? A Qualitative Usability Study. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), Dallas, TX, USA, 30 October–3 November 2017; pp. 311–328.
18. Acar, Y.; Backes, M.; Fahl, S.; Kim, D.; Mazurek, M.; Stransky, C. You Get Where You're Looking for the Impact of Information Sources on Code Security. In Proceedings of the 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2016; IEEE: San Jose, CA, USA, 2016; pp. 289–305.
19. Felix, F.; Konstantin, B.; Huang, X.; Christian, S.; Yasemin, A.; Michael, B.; Sascha, F. Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; IEEE: San Jose, CA, USA, 2017; pp. 121–136.
20. Yasemin, A.; Michael, B.; Sascha, F.; Doowon, K.; Michelle, M.; Christian, S. How Internet Resources Might Be Helping You Develop Faster but Less Securely. *IEEE Secur. Priv.* **2017**, *15*, 50–60.
21. Imai, H.; Kanaoka, A. Time Series Analysis of Copy-and-Paste Impact on Android Application Security. In Proceedings of the 2018 13th Asia Joint Conference on Information Security (AsiaJCIS), Guilin, China, 8–9 August 2018; pp. 15–22.

22. Charles, W.; Awais, R.; Noble, J. Reaching the Masses: A New Subdiscipline of App Programmer Education. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016), Seattle, WA, USA, 13–18 November 2016; ACM: Seattle, WA, USA, 2016; pp. 936–939.

23. Boopathi, K.; Sreejith, S.; Bithin, A. Learning Cyber Security Through Gamification. *Indian J. Sci. Technol.* **2015**, *8*, 642.

24. Tillmann, N.; de Halleux, J.; Xie, T.; Bishop, J. Code hunt: Gamifying teaching and learning of computer science at scale. In Proceedings of the First ACM Conference on Learning @ Scale Conference (L@S '14), Atlanta, GA, USA, 4–5 March 2014; ACM: Atlanta, Georgia, USA, 2014; pp. 221–222.

25. Denning, T.; Lerner, A.; Shostack, A.; Kohno, T. Control-Alt-Hack: The Design and Evaluation of a Card Game for Computer Security Awareness and Education. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13), Berlin, Germany, 4–8 November 2013; ACM: Berlin, Germany, 2013; pp. 915–928.

26. Rieff, I. Systematically Applying Gamification to Cyber Security Awareness Trainings: A framework and case study approach. Master's Thesis, Faculty of Technology, Policy and Management, Delft, The Netherlands, 2018.

27. Matthew, G.; Matthew, S. Developers Are Not the Enemy! The Need for Usable Security APIs. *IEEE Secur. Priv.* **2016**, *14*, 40–46.

28. Lo Iacono, L.; Gorski, P.L. I Do and I Understand. Not Yet True for Security APIs. So Sad. In Proceedings of the 2nd European Workshop on Usable Security, Internet Society, Paris, France, 29 April 2017.

29. Mosqueira-Rey, E.; Alonso-Ríos, D.; Moret-Bonillo, V.; Fernández-Varela, I.; Álvarez Estévez, D. A Systematic Approach to API Usability: Taxonomy-Derived Criteria and a Case Study. *Inf. Softw. Technol.* **2018**, *97*, 46–63.

30. O'Callaghan, P. The API Walkthrough Method: A Lightweight Method for Getting Early Feedback about an API. In *PLATEAU '10: Evaluation and Usability of Programming Languages and Tools*; Association for Computing Machinery: New York, NY, USA, 2010.

31. Antonopoulos, A.M. *Mastering Bitcoin: Programming the Open Blockchain*, 2nd ed.; OCLC: 988250213; O'Reilly: Sebastopol, CA, USA, 2017.

32. Yli-Huumo, J.; Ko, D.; Choi , S.; Park, S.; Smolander, K. Where Is Current Research on Blockchain Technology? A Systematic Review. *PLoS ONE* **2016**, *10*, e0163477.

33. Meva, D.D. Issues and Challenges with Blockchain: A Survey. *Int. J. Comput. Sci. Eng.* **2018**, *6*, 488–491.

34. Tikhomirov, S.; Voskresenskaya, E.; Ivanitskiy, I.; Takhaviev, R.; Marchenko, E.; Alexandrov, Y. SmartCheck: Static Analysis of Ethereum Smart Contracts. In Proceedings of the 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Gothenburg, Sweden, 27 May–3 June 2018; pp. 9–16.

35. Feist, J.; Grieco, G.; Groce, A. Slither: A Static Analysis Framework for Smart Contracts. In Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), Montreal, QC, Canada, 27 May 2019.

36. Ye, J.; Ma, M.; Peng, T.; Xue, Y. A Software Analysis Based Vulnerability Detection System For Smart Contracts. In *Integrating Research and Practice in Software Engineering*; Jarzabek, S., Poniszewska-Marańda, A., Madeyski, L., Eds.; Springer: Cham, Switzerland, 2020; pp. 69–81.

37. Grishchenko, I.; Maffei, M.; Schneidewind, C. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In *Computer Aided Verification*; Chockler, H., Weissenbacher, G., Eds.; Springer: Cham, Switzerland, 2018; pp. 51–78.

38. Liu, J.; Liu, Z. A Survey on Security Verification of Blockchain Smart Contracts. *IEEE Access* **2019**, *7*, 77894–77904.

39. Wijayarathna, C.; Arachchilage, N.A.G. Am I Responsible for End-User's Security? A Programmer's Perspective. *arXiv* **2018**, arXiv:1808.01481.

40. Vassallo, C.; Panichella, S.; Palomba, F.; Proksch, S.; Zaidman, A.; Gall, H.C. Context is king: The developer perspective on the usage of static analysis tools. In Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Campobasso, Italy, 20–23 March 2018; pp. 38–49.

41. Murphy-Hill, E.; Sadowski, C.; Head, A.; Daughtry, J.; Macvean, A.; Jaspan, C.; Winter, C. Discovering API Usability Problems at Scale. In Proceedings of the 2nd International Workshop on API Usage and Evolution (WAPI '18), Gothenburg, Sweden, 2–4 June 2018; ACM: New York, NY, USA, 2018; pp. 14–17.

42.   Paletov, R.; Tsankov, P.; Raychev, V.; Vechev, M. Inferring Crypto API Rules from Code Changes. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018), Philadelphia, PA, USA, 18–22 June 2018; ACM: New York, NY, USA, 2018; pp. 450–464.

43.   Oliveira, D.S.; Lin, T.; Rahman, M.S.; Akefirad, R.; Ellis, D.; Perez, E.; Bobhate, R.; DeLong, L.A.; Cappos, J.; Brun, Y. API Blindspots: Why Experienced Developers Write Vulnerable Code. In Proceedings of the Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018), Baltimore, MD, USA, 12–14 August 2018; USENIX Association: Baltimore, MD, USA, 2018; pp. 315–328.

44.   Wijayarathna, C.; Arachchilage, N.A.G. Why Johnny Can't Store Passwords Securely? A Usability Evaluation of Bouncycastle Password Hashing. In Proceedings of the 22nd International Conference on Evaluation and Assessment in Software Engineering (EASE'18), Christchurch, New Zealand, 28–29 June 2018; pp. 205–210.

45.   Mindermann, K.; Wagner, S.   Usability and Security Effects of Code Examples on Crypto APIs—CryptoExamples: A platform for free, minimal, complete and secure crypto examples. *CoRR* **2018**, abs/1807.01095.

46.   Mindermann, K.; Keck, P.; Wagner, S. How Usable Are Rust Cryptography APIs? In Proceedings of the 2018 IEEE International Conference on Software Quality, Reliability and Security (QRS), Lisbon, Portugal, 16–20 July 2018; pp. 143–154.

47.   Krüger, S.; Nadi, S.; Reif, M.; Ali, K.; Mezini, M.; Bodden, E.; Göpfert, F.; Günther, F.; Weinert, C.; Demmler, D.; et al. CogniCrypt: Supporting Developers in Using Cryptography. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana-Champaign, IL, USA, 30 October–3 November 2017; pp. 931–936.

48.   Pieczul, O.; Foley, S.; Zurko, M.E.   Developer-Centered Security and the Symmetry of Ignorance. In Proceedings of the 2017 New Security Paradigms Workshop on ZZZ (NSPW), Santa Cruz, CA, USA, 1–4 October 2017; pp. 46–56.

49.   Sam, W.; Michael, C.; Brad, M.; Jonathan, A.; Joshua, S. Empirical Studies on the Security and Usability Impact of Immutability. In Proceedings of the 2017 IEEE Cybersecurity Development (SecDev), Cambridge, MA, USA, 24–26 September 2017; IEEE: Cambridge, MA, USA, 2017; pp. 50–53.

50.   Sarah, N.; Stefan, K.; Mira, M.; Eric, B. "Jumping Through Hoops": Why do Java Developers Struggle with Cryptography APIs? In Proceedings of the 38th International Conference on Software Engineering (ICSE '16), Austin, TX, USA, 14–22 May 2016; IEEE: Austin, TX, USA, 2016; pp. 935–946.

51.   Mindermann, K. Are Easily Usable Security Libraries Possible and How Should Experts Work Together to Create Them? In Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '16), Austin, TX, USA, 14–22 May 2016; pp. 62–63.

52.   Medeiros, I.; Neves, N.; Correia, M. DEKANT: A Static Analysis Tool That Learns to Detect Web Application Vulnerabilities. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA), Saarbrücken, Germany, 18–20 July 2016; pp. 1–11.

53.   Ma, S.; Lo, D.; Li, T.; Deng, R.H. CDRep: Automatic Repair of Cryptographic Misuses in Android Applications. In Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS '16), Xi'an, China, 30 May–3 June 2016; pp. 711–722.

54.   Indela, S.; Kulkarni, M.; Nayak, K.; Dumitraş, T. Helping Johnny Encrypt: Toward Semantic Interfaces for Cryptographic Frameworks. In Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Amsterdam, The Netherlands, 2–4 November 2016; ACM: New York, NY, USA, 2016; pp. 180–196.

55.   Indela, S.; Kulkarni, M.; Nayak, K.; Dumitraş, T. Toward Semantic Cryptography APIs. In Proceedings of the 2016 IEEE Cybersecurity Development (SecDev), Boston, MA, USA, 3–4 November 2016; pp. 9–14.

56.   Gorski, P.L.; Iacono, L.L. Towards the Usability Evaluation of Security APIs. In Proceedings of the Tenth International Symposium on Human Aspects of Information Security & Assurance (HAISA 2016), Frankfurt, Germany, 19–21 July 2016.

57.   Asaduzzaman, M.; Roy, C.K.; Schneider, K.A.; Hou, D. CSCC: Simple, Efficient, Context Sensitive Code Completion.   In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; pp. 71–80.

58.   Das, S.; Gopal, V.; King, K.; Venkatraman, A. *IV = 0 Security Cryptographic Misuse of Libraries*; MIT Computer Science and Artificial Intelligence Laboratory: Cambridge, MA, USA, 2014.

59. Egele, M.; Brumley, D.; Fratantonio, Y.; Kruegel, C. An Empirical Study of Cryptographic Misuse in Android Applications. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13), Berlin, Germany, 4–8 October 2013; pp. 73–84.

60. Weir, C.; Rashid, R.; Noble, J. How to Improve the Security Skills of Mobile App Developers: Comparing and Contrasting Expert Views. In *Symposium on Usable Privacy and Security (SOUPS)*; USENIX Association: Denever, CO, USA, 2016; pp. 1–7.

61. Yasemin, A.; Christian, S.; Dominik, W.; Charles, W.; Michelle, M.; Sascha, F. Developers Need Support, Too: A Survey of Security Advice for Software Developers. In Proceedings of the 2017 IEEE Cybersecurity Development (SecDev), Cambridge, MA, USA, 24–26 September 2017; IEEE: Cambridge, MA, USA, 2017; pp. 22–26.

62. Weir, C.; Rashid, A.; Noble, J. I'd Like to Have an Argument, Please Using Dialectic for Effective App Security. In Proceedings of the 2nd European Workshop on Usable Security (EuroUSEC 2017), Paris, France, 29 April 2017; Internet Society: Reston, VA, USA, 2017.

63. Sodhi, B.; Sharma, S. Using Stack Overflow content to assist in code review. *CoRR* **2018**, abs/1803.05689.

64. Lopez, T.; Tun, T.; Bandara, A.; Nuseibeh, B.; Sharp, H.; Levine, M. An Investigation of Security Conversations in Stack Overflow: Perceptions of Security and Community Involvement. In Proceedings of the First International Workshop on Security Awareness from Design to Deployment (SEAD'18), Gothenburg, Sweden, 27 May–3 June 2018; ACM: New York, NY, USA, 2018.

65. Meng, N.; Nagy, S.; Yao, D.; Zhuang, W.; Arango-Argoty, G. Secure Coding Practices in Java: Challenges and Vulnerabilities. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 18–26 May 2018; pp. 372–383.

66. Tabassum, M.; Watson, S.; Lipford, H.R. *Comparing Educational Approaches to Secure Programming: Tool vs. TA*; USENIX Association: Santa Clara, CA, USA, 2017.

67. Rebecca, B.; Abigail, M.; Jialiu, L.; Jason, H.; Lorrie, C. The Privacy and Security Behaviors of Smartphone App Developers. In Proceedings of the Workshop on Usable Security (USEC'14), San Diego, CA, USA, 23 February 2014; pp. 1–10.

68. Poller, A.; Kocksch, L.; Turpe, S.; Epp, F.; Kinder-Kurlanda, K. Can Security Become a Routine?: A Study of Organizational Change in an Agile Software Development Group. In Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17), Portland, OR, USA, 25 February–1 March 2017; pp. 2489–2503.

69. Matthiesen, S.; Bjørn, P.; Petersen, L. "Figure Out How to Code with the Hands of Others": Recognizing Cultural Blind Spots in Global Software Development. In Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW '14), Baltimore, MD, USA, 15–19 February 2014; pp. 1107–1119.

70. Ponemon. *The State of Mobile Application Insecurity*; Survey; Ponemon Institute LLC: Traverse, MI, USA, 2015.

71. Assal, H.; Chiasson, S. "Think Secure from the Beginning": A Survey with Software Developers. In Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19), Glasgow, Scotland, UK, 4–9 May 2019; ACM: New York, NY, USA, 2018; pp. 1–13.

72. Rebecca, B.; Lorrie, C. Improving App Privacy: Nudging App Developers to Protect User Privacy. *IEEE Secur. Priv.* **2014**, *12*, 55–58.

73. Zhu, J.; Lipford, H.; Chu, B. Supporting secure programming in web applications through interactive static analysis. *J. Adv. Res.* **2014**, *5*, 449–462.

74. Oleksii, K.; Olga, B.; Michael, G. Code review quality: How developers see it. In Proceedings of the 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, USA, 14–22 May 2016; pp. 1028–1038.

75. Sascha, F.; Marian, H.; Henning, P.; Markus, K.; Matthew, S. Rethinking SSL Development in an Appified World. In Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security (CCS '13), Berlin, Germany, 4–8 November 2013; pp. 49–60.

76. Thomas, T.W.; Tabassum, M.; Chu, B.; Lipford, H. Security During Application Development: An Application Security Expert Perspective. In Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18), Montreal, QC, Canada, 21–26 April 2018; ACM: New York, NY, USA, 2018; pp. 262:1–262:12.

77. Grance, T.; Hash, J.; Stevens, M. *Security Considerations in the Information System Development Life Cycle*, rev. 1 ed.; Number 800-64 in Special Publication; National Institute of Standards and Technology (NIST): Gaithersburg, MD, USA, 2004.

78. Siponen, M.; Baskerville, R.; Kuivalainen, T. Integrating Security into Agile Development Methods. In Proceedings of the 38th Annual Hawaii International Conference on System Sciences, Big Island, HI, USA, 3–6 January 2005; pp. 185a.

79. Michael, H.; Steve, L. *The Security Development Lifecycle*; Microsoft Press: Redmond, WA, USA, 2006; Volume 8.

80. Geer, D. Are Companies Actually Using Secure Development Life Cycles? *Computer* **2010**, *43*, 12–16.

81. Baca, D.; Carlsson, B. Agile Development with Security Engineering Activities. In Proceedings of the 2011 International Conference on Software and Systems Process (ICSSP '11), Honolulu, HI, USA, 21–22 May 2011; ACM: New York, NY, USA, 2011; pp. 149–158.

82. Sonia; Singhal, A. Development of Agile Security Framework Using a Hybrid Technique for Requirements Elicitation. In *Advances in Computing, Communication and Control*; Unnikrishnan, S., Surve, S., Bhoir, D., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 178–188.

83. Assal, H.; Chiasson, S. Motivations and Amotivations for Software Security. In Proceedings of the USENIX Symposium on Usable Privacy and Security (SOUPS 2018), Baltimore, MD, USA, 12–14 August 2018.

84. Assal, H.; Chiasson, S. Security in the Software Development Lifecycle. In Proceedings of the Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018), Baltimore, MD, USA, 12–14 August 2018; pp. 281–296.

85. Nicolas, B. Studying the Impact of Developer Communication on the Quality and Evolution of a Software System A Doctoral Dissertation Retrospective. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; IEEE: Victoria, BC, Canada, 2014; pp. 651–656.

86. Zhu, J.; Xie, J.; Lipford, H.; Chu, B. Evaluating interactive support for secure programming. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '12), Austin, TX, USA, 5–10 May 2012; ACM: Austin, TX, USA, 2012; pp. 2707–2716.

87. Martin, U.; Vashek, M. Why Johnny the Developer Can't Work with Public Key Certificates An Experimental Study of OpenSSL Usability. In *CT-RSA 2018*; Lecture Notes in Computer Science (LNCS); Springer: San Francisco, CA, USA, 2018; Volume 10808, pp. 45–64.

88. Zinaida, B.; Gabriele, L.; Daniela, O.; Simon, P.; Sven, U. Maybe Poor Johnny Really Cannot Encrypt: The Case for a Complexity Theory for Usable Security. In Proceedings of the 2015 New Security Paradigms Workshop (NSPW '15), Twente, The Netherlands, 8–11 September 2015; pp. 85–99.

89. Bernstein, D.J.; Lange, T.; Schwabe, P. The Security Impact of a New Cryptographic Library. In *Progress in Cryptology—LATINCRYPT 2012*; Hevia, A., Neven, G., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 159–176.

90. Gorski, P.L.; Iacono, L.L.; Wermke, D.; Stransky, C.; Möller, S.; Acar, Y.; Fahl, S. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In Proceedings of the Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018), Baltimore, MD, USA, 12–14 August 2018; pp. 265–281.

91. Muhammad, A.; Chanchal, R.; Kevin, S.; Daqing, H. Context-Sensitive Code Completion Tool for Better API Usability. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; IEEE: Victoria, BC, Canada, 2014; pp. 622–624.

92. Chamila, W.; authorNalin, A.; Jill, S. A Generic Cognitive Dimensions Questionnaire to Evaluate the Usability of Security APIs. In *HAS 2017*; Lecture Notes in Computer Science (LNCS); Springer: Vancouver, BC, Canada, 2017; Volume 10292, pp. 160–173.

93. Yasemin, A.; Michael, B.; Sascha, F.; Simson, G. Comparing the Usability of Cryptographic APIs. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 154–171.

94. DeTreville, J. Binder, a logic-based security language. In Proceedings of the 2002 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 12–15 May 2002; pp. 105–113.

95. Nielsen, J.D.; Schwartzbach, M.I. A Domain-specific Programming Language for Secure Multiparty Computation. In Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security (PLAS '07), San Diego, CA, USA, 14 June 2007; ACM: New York, NY, USA, 2007; pp. 21–30.

96. Jagomägis, R. SecreC: A Privacy-Aware Programming Language with Applications in Data Mining. Master's Thesis, University of Tartu, Tartu, Estonia, 2010.

97. Mettler, A.; Wagner, D.; Close, T. *Joe-E: A Security-Oriented Subset of Java*; Report; Network and Distributed Systems Symposium, Internet Society: San Diego, CA, USA, 2010.

98. Foster, N.; Harrison, R.; Freedman, M.J.; Monsanto, C.; Rexford, J.; Story, A.; Walker, D. Frenetic: A Network Programming Language. *SIGPLAN Not.* **2011**, *46*, 279–291.

99. Ligia, N.; Darya, K.; Stephanie, B.; Benjamin, C.; Alex, P.; Jonathan, A. Wyvern: A Simple, Typed, and Pure Object-Oriented Language. In Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI'13), Montpellier, France, 1–5 July 2013.

100. Matsakis, N.D.; Klock, II, F.S. The Rust Language. *Ada Lett.* **2014**, *34*, 103–104.

101. Darya, K.; Alex, P.; Jonathan, A. Wyvern: Impacting Software Security via Programming Language Design. In Proceedings of the 5th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU'14), Portland, OR, USA, 20–24 October 2014; pp. 57–58.

102. Yang, J.; Yessenov, K.; Solar-Lezama, A. A Language for Automatically Enforcing Privacy Policies. *SIGPLAN Not.* **2012**, *47*, 85–96.

103. Naim, R.; Nizam, M.F.; Hanamasagar, S.; Noureddine, J. *Comparative Studies of 10 Programming Languages within 10 Diverse Criteria*; Term Report COMP6411-S10; Concordia University: Montreal, ON, Canada, 2010.

104. Turner, S. Security vulnerabilities of the top ten programming languages: C, Java, C++, Objective-C, C#, PHP, Visual Basic, Python, Perl, and Ruby. *J. Technol. Res.* **2014**, *5*, 1–16.

105. Hoar, C.A.R. *Hints on Programming Language Design*; Technical Report; Stanford University: Stanford, CA, USA, 1974; Volume 20.

106. Cass, S.; Bulusu, P. *Interactive: The Top Programming Languages 2018*; Technical Report; IEEE: Piscataway, NJ, USA, 2018.

107. Colin, H. *Security in Programming Languages*; Tufts University: Medford, MA, USA, 2015.

108. Rahul, G.; Carlos, J.; Alex, G. Code coverage for suite evaluation by developers. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Hyderabad India, 31 May–7 June 2014; pp. 72–82.

109. Vestola, M. Evaluating and Enhancing FindBugs to Detect Bugs from Mature Software; Case Study in Valuatum. Master's Thesis, Aalto University, Espoo, Finland, 2012.

110. Daniel, V.; Rock, S.; Elissa, R.; Jeremy, H.; Michelle, M. Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes. In Proceedings of the 2018 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 20–24 May 2018; IEEE: San Francisco, CA, USA, 2018; pp. 134–151.

111. Vincent, B.; Anne, E.; Nicolas, A.; Sylvain, C.; Pascal, C.; Stéphane, D. What are the Testing Habits of Developers? A Case Study in a Large IT Company. In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; IEEE: Shanghai, China, 2017; pp. 58–68.

112. Mario, L.V.; Cárlos, B.C.; Kevin, M.; Denys, P. How do Developers Test Android Applications? In Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 17–22 September 2017; IEEE: Shanghai, China, 2017; pp. 613–622.

113. Riccardo, S.; James, W.; Wouter, J. Static Analysis Versus Penetration Testing: A Controlled Experiment. In Proceedings of the 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), Pasadena, CA, USA, 4–7 November 2013; IEEE: Pasadena, CA, USA, 2013; pp. 451–460.

114. Enck, W.; Octeau, D.; McDaniel, P.; Swarat, C. A Study of Android Application Security. In Proceedings of the 20th USENIX conference on Security (SEC'11), San Francisco, CA, USA, 8–12 August 2011; pp. 20–21.

115. Smith, J.; Johnson, B.; Murphy-Hill, E.; Chu, B.; Richter, H. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Trans. Softw. Eng.* **2018**, doi:10.1109/TSE.2018.2810116.

116. Jovanovic, N.; Kruegel, C.; Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06), Berkeley/Oakland, CA, USA, 21–24 May 2006; IEEE: Berkeley/Oakland, CA, USA, 2006; pp. 258–263.

117. Dehlinger, J.; Feng, Q.; Hu, L. SSVChecker: Unifying Static Security Vulnerability Detection Tools in an Eclipse Plug-in. In Proceedings of the 2006 OOPSLA Workshop on Eclipse Technology eXchange (eclipse '06), New York, NY, USA, 22–23 October 2006; pp. 30–34.

118. Pérez, P.M.; Filipiak, J.; Sierra, J.M. LAPSE+ Static Analysis Security Software: Vulnerabilities Detection in Java EE Applications. In *Future Information Technology*; Park, J.J., Yang, L.T., Lee, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 148–156.

119. Li, K.; Hebert, C.; Lindemann, J.; Sauter, M.; Mack, H.; Schröer, T.; Tiple, A. Tool support for secure programming by security testing. In Proceedings of the 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Graz, Austria, 13–17 2015; pp.1–4.

120. Sven, A.; Sebastian, P.; Sarah, N. FeedBaG: An interaction tracker for Visual Studio. In Proceedings of the 2016 IEEE 24th International Conference on Program Comprehension (ICPC), Austin, TX, USA, 16–17 May 2016; IEEE: Austin, TX, USA, 2016; pp. 1–3.

121. Sebastian, P.; Sarah, N.; Sven, A.; Mira, M. Enriching in-IDE process information with fine-grained source code history. In Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 20–24 February 2017; IEEE: Klagenfurt, Austria, 2017; pp. 250–260.

122. Duc, N.; Dominik, W.; Yasemin, A.; Michael, B.; Charles, W.; Sascha, F. A Stitch in Time: Supporting Android Developers in writing secure Code. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), Dallas, TX, USA, 2017; pp. 1065–1077.

123. Baset, A.Z.; Denning, T. IDE Plugins for Detecting Input-Validation Vulnerabilities. In Proceedings of the 2017 IEEE Security and Privacy Workshops (SPW), San Jose, CA, USA, 25 May 2017; IEEE: San Jose, CA, USA, 2017; pp. 143–146.

124. Shen, H.; Fang, J.; Zhao, J. EFindBugs: Effective Error Ranking for FindBugs. In Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, Berlin, Germany, 21–25 March 2011; pp. 299–308.

125. Xiao, S.; Witschey, J.; Murphy-Hill, E. Social Influences on Secure Development Tool Adoption: Why Security Tools Spread. In Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW '14), Baltimore, MD, USA, 15–19 February 2014; ACM: Baltimore, MD, USA, 2014; pp. 1095–1106.

126. Jim, W.; Olga, Z.; Allaire, W.; Emerson, M.H.; Chris, M.; Thomas, Z. Quantifying Developers' Adoption of Security Tools. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), Bergamo, Italy, 30 August–30 September 2015; pp. 260–271.

127. Brittany, J.; Yoonki, S.; Emerson, M.H.; Robert, B. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13), San Francisco, CA, USA, 18–26 May 2013; pp. 672–681.

128. Maria, C.; Christian, B. What Developers Want and Need from Program Analysis: An Empirical Study. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering ASE, Singapore, 3–7 September 2016; pp. 332–343.

129. Goseva-Popstojanovaa, K.; Perhinschi, A. On the capability of static code analysis to detect security vulnerabilities. *Inf. Softw. Technol.* **2015**, *68*, 18–33.

130. Ruthruff, J.R.; Penix, J.; Morgenthaler, J.D.; Elbaum, S.; Rothermel, G. Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach. In Proceedings of the 30th International Conference on Software Engineering (ICSE '08), Leipzig, Germany, 10–18 May 2008; ACM: Leipzig, Germany, 2008; pp. 341–350.

131. Heckman, S.; Williams, L. A comparative evaluation of static analysis actionable alert identification techniques. In Proceedings of the 9th International Conference on Predictive Models in Software Engineering (PROMISE '13), Baltimore, MD, USA, 9 October 2013.

132. Yoon, J.; Jin, M.; Jung, Y. Reducing False Alarms from an Industrial-Strength Static Analyzer by SVM. In Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference, Jeju, Korea, 1–4 December 2014;Volume 2, pp. 3–6.

133. Perl, H.; Dechand, S.; Smith, M.; Arp, D.; Yamaguchi, F.; Rieck, K.; Fahl, S.; Acar, Y. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15), Denver, CO, USA, 12–16 October 2015; ACM: Denver, CO, USA, 2015; pp. 426–437.

134. Kristín, T.; Mauricio, A.; Arie, v.D. Why and how JavaScript developers use linters. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, (ASE 2017), Urbana, IL, USA, 30 October–3 November 2017; pp. 578–589.

135. Kevin, L.; Titus, B.; Emerson, M.H. Can Social Screencasting Help Developers Learn New Tools. In Proceedings of the 2015 IEEE/ACM 8th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '15), Florence, Italy, 18 May 2015; IEEE Press Piscataway: Florence/Firenze, Italy, 2015; pp. 113–114.

136. Fabio, P.; Gabriele, B.; Massimiliano, D.P.; Rocco, O.; Andrea, D.L. Do they Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, 29 September–3 October 2014; IEEE: Victoria, BC, Canada, 2014; pp. 101–110.

137. Lavallée, M.; Robillard, P. Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; IEEE: Florence, Italy, 2015; pp. 677–687.

138. Xie, J.; Lipford, H.; Chu, B. Why do programmers make security errors? In Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), Pittsburgh, PA, USA, 18–22 September 2011; IEEE: Pittsburgh, PA, USA, 2011; pp. 161–164.

139. Titus, B.; Justin, S.; Kevin, L.; Elisabeth, H.; Jing, F.; Emerson, M.H.; Chris, P. Do Developers Read Compiler Error Messages? In *Proceedings of the 39th International Conference on Software Engineering*; IEEE Press Piscataway: Buenos Aires, Argentina, 2017; pp. 575–585.

140. Samim, M.; Chris, P. Can Automated Pull Requests Encourage Software Developers to Upgrade Out-of-Date Dependencies? In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017), Urbana-Champaign, IL, USA, 30 October–3 November 2017; pp. 84–94.

141. Felipe, E.; Fernando, C. A Study on Developers' Perceptions about Exception Handling Bugs. In Proceedings of the 2013 IEEE International Conference on Software Maintenance (ICSM), Eindhoven, The Netherlands, 22–28 September 2013; IEEE: Eindhoven, The Netherlands, 2013; pp. 448–451.

142. Raula, K.; Daniel, G.; Ali, O.; Takashi, I.; Katsuro, I. Do developers update their library dependencies? *Empir. Softw. Eng.* **2018**, *23*, 384–417.

143. Wijayarathna, C.; Arachchilage, N.A.G. A methodology to Evaluate the Usability of Security APIs. In Proceedings of the 2018 IEEE International Conference on Information and Automation for Sustainability (ICIAfS), Colombo, Sri Lanka, 21–22 December 2018; pp. 1–6.

144. Grill, T.; Polacek, O.; Tscheligi, M. Methods towards API Usability: A Structural Analysis of Usability Problem Categories. In *Human-Centered Software Engineering*; Winckler, M., Forbrig, P., Bernhaupt, R., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 164–180.

145. Nielsen, J. *10 Usability Heuristics for User Interface Design*; Technical Report; Academic Press: Boston, MA, USA, 1994.

146. Parr, T. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*; The Pragmatic Programmers; Cambridge University Press: Cambridge, UK, 2010; OCLC: Ocn419869921.

147. Tschannen, P.; Ahmed, A. On the Evaluation of the Security Usability of Bitcoin's APIs. In Proceedings of the Evaluation and Assessment in Software Engineering (EASE '20), Trondheim, Norway, 15–17 April 2020; Association for Computing Machinery: New York, NY, USA, 2020; pp. 405–412. doi:10.1145/3383219.3383277.

148. Joel, G.; Philip, R.; Rastislav, B.; Björn, H.; Koushik, S. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), Hyderabad India, 31 May–7 June 2014; pp. 653–663.

149. Chin, E.; Felt, A.P.; Greenwood, K.; Wagner, D. Analyzing inter-application communication in Android. In Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services, Bethesda, MD, USA, 28 June–1 July 2011; ACM: Bethesda, MD, USA, 2011; pp. 239–252.

150. Youn, L.; Peera, Y.; Arman, S.; Daye, N.; Nenad, M. SEALANT: A Detection and Visualization Tool for Inter-App Security Vulnerabilities in Android. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 30 October–3 November 2017; pp. 883–888.

151. Algaith, A.; Nunes, P.; Fonseca, J.; Gashi, I.; Vieira, M. In Proceedings of the Finding SQL Injection and Cross Site Scripting Vulnerabilities with Diverse Static Analysis Tools, Iasi, Romania, 10–14 September 2018.

152. Smith, R. *Working Draft, Standard for Programming Language C++*; N4296; Google Inc.: Mountain View, CA, USA, 2017

153. Sutter, H.; Alexandrescu, A. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*; Addison-Wesley Professional: Boston, MA, USA, 2004.

154. Gabriel Dos Reis, J.; Daniel García, F.L.M.F.; Lahiri, S. *Simple Contracts for C++*; N4415, Online 2015. Available online: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4415.pdf (accessed on 30 June 2020).

155. Kochhar, P.S.; Lo, D. Revisiting Assert Use in GitHub Projects. In Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering (EASE'17), Copenhagen, Denmark, 15–17 April 2017; ACM: New York, NY, USA, 2017; pp. 298–307.