Being the Developers' Friend: Our Experience Developing a High-Precision Tool for Secure Coding

Danfeng (Daphne) Yao | Virginia Tech
Sazzadur Rahaman | University of Arizona
Ya Xiao, Sharmin Afrose, and Miles Frantz | Virginia Tech
Ke Tian | Palo Alto Networks
Na Meng | Virginia Tech
Cristina Cifuentes, Yang Zhao, Nicholas Allen, and Nathan Keynes | Oracle Labs Australia
Barton P. Miller and Elisa Heymann | University of Wisconsin–Madison
Murat Kantarcioglu | University of Texas at Dallas
Fahad Shaon | Data Security Technologies

We discuss the needs and challenges of deployable security research by sharing our experience designing CryptoGuard, a high-precision tool for detecting cryptographic application programming interface misuses. Our project has produced multiple benchmarks as well as measurement results on state-of-the-art solutions.

In summer 2017, we were shocked to see several highly influential but misleading StackOverflow (SO) posts about Java security coding. These posts included suggestions such as disabling cross-site request forgery (CSRF) tokens, accepting all certificates with an empty verification method, and using obsolete hash functions to get rid of compiling errors.[1] Virtually all of them are marked as accepted answers by the question askers. Collectively, 17 problematic SO posts were viewed 622,922 times as of August 2017, as shown in Table 1. The high number of views is expected as SO estimates that 21 million site visitors are professional developers and university-level students preparing for coding careers.[2] Some SO code makes its way into mobile devices—15.4% of apps contain code snippets copied from SO, and most of them contain at least one insecure code snippet.[3] Additionally, we also observed cyberbullying behaviors on SO, where a highly reputed person dismissed the secure answer offered by an SO user with a low reputation score. Such cyberbullying behaviors, if they widely exist, could discourage people from providing technically correct, sound, and useful information about secure coding.[1]

From the aforementioned SO posts, we, as security researchers, observed a disconnect between security principles and coding practices. Although a security researcher may believe that the CSRF problem has long been eradicated, the SO technical forum evidences the opposite. An SO user stated, "Adding csrf().disable() solved the issue!!! I have no idea why it was enabled by default." CSRF was a high-profile discovery in 2007. Since then, this attack and its defenses (including using validation tokens and examining referrer headers) have routinely been taught in university cybersecurity classes. CSRF, also known as a *confused deputy problem*, is a web attack where the victim's cached credential (e.g., session ID with a bank) is automatically appended to outgoing requests by the victim's browser, even when the requests are forged by attackers, resulting in the attacker's forged requests being successfully authenticated and processed as the victim's own. Attackers stealing money from a

Digital Object Identifier 10.1109/MSEC.2022.3159481
Date of current version: 26 May 2022

victim's online bank account and changing a victim's email password are just two examples of possible consequences. A popular third-party security framework, Spring Security, provides built-in CSRF protection and enables protection by default. This protection mechanism requires the framework users (e.g., developers of web applications) to include CSRF tokens when sending state-modification HTTP requests. If a developer wants to avoid the inconvenience of creating or using CSRF tokens, they can call "csrf().disable()" to bypass the protection and destroy the security promise by Spring Security.

Some users post vulnerable code due to conceptual misunderstandings. For example, one SO user stated, "I want my client to accept any certificate (because I'm only ever pointing to one server)." This statement was made in the context of public-key certificate verification, likely for a specialized app developed to connect to its back-end server. However, this ever-pointing-to-one-server concept is troubling because without certificate verification, attacks such as Domain Name System poisoning, phishing apps, and man-in-the-middle attacks can easily break end-to-end security. Challenges in coding certificate verification have also been widely reported by researchers.

Vulnerable code is also making its way into widely used security software, impacting cryptographic functions. For instance, in December 2015, Willem Pinckaers reported on Twitter about the global variable reuse in Juniper Networks' Dual Elliptic Curve Deterministic Random Bit Generator.[4] This coding error impacted an important loop variant, resulting in randomization operations not being executed at all, which resulted in exploitable vulnerabilities in network devices.[5] In 2017, Equifax reported a data breach incident that impacted 147 million people partly because of an expired digital credential.

From the technical discussion by developers on online forums, we perceived a strong message: developers need help. They need help with writing secure code and understanding the security implications underneath the coding options.

To help with secure coding, we designed CryptoGuard, which focuses on a specific category of vulnerabilities, that being the application programming interface (API) misuses of two Java libraries: Java Cryptography Architecture (JCA) and Java Secure Socket Extension (JSSE). We made this design choice for two reasons. First, Java developers typically use the APIs offered by both libraries to ease their cryptographic implementation, but misusing some of the APIs can make the resulting implementation exploitable by attackers. Second, our prior work[1] and studies by other researchers[3] all show that many developers misunderstand or misuse the APIs of JCA and JSSE and need help in understanding the security implications of alternative API options and addressing API-misuse issues.

Table 1. The view numbers of 17 influential insecure posts on the SO forum.[1]

Topic of posts	Total number of views	Number of posts	Minimum number of views	Maximum number of views
Disable CSRF protection	39,863	5	261	28,183
Trust all certificates	491,567	9	95	391,464
Obsolete hash functions	91,492	3	1,897	86,070
Total	**622,922**	**17**	—	—

[1]*The numbers were collected in August 2017.*

Why Are Some API Misuses Difficult to Detect?

Although string matching can detect occurrences of insecure parameters (e.g., uses of insecure cryptographic hash functions MD5 and SHA1, weak block cipher Data Encryption Standard, or small key sizes), complications may arise in large software projects.

Tracking variable values across method invocation is one challenge, where API arguments are instantiated elsewhere in a different method or class. Thus, to determine whether arguments of a cryptographic API are secure, one needs to track data flows (e.g., definition-and-use relationships) across procedure boundaries. Static, interprocedural data flow analysis forms the basis of our detection. Specifically, we utilize interprocedural backward slicing to collect program statements (known as the *program slice*[6]) that impact an element of interest (referred to as the *slicing criterion*) and then scan the slices for misuse patterns.

Hasn't Static, Interprocedural Dataflow Analysis Been Known for Decades?

Yes; however, specializing and customizing general-purpose static analysis techniques (such as Soot, a static program analysis framework) for specific security detection requires much innovation. One challenge is the semantic gap, the difference in meaning between cryptography and code. An important process in our CryptoGuard work is to determine exactly what to look for in programs.

We need to map abstract cryptographic concepts to concrete programming elements and data flow

Table 2. CryptoGuard's precision under categories of cryptographic vulnerabilities excluding the category on untrusted pseudorandom number generator (PRNG), when evaluating 46 Apache projects.[7]

Vulnerability types	Number of alerts	Number of true positives	Precision (%)
1 and 2. Predictable keys	264	248	94.1
3. Hardcoded KeyStore password	148	148	100
4. Dummy hostname verifier	12	12	100
5. Dummy certification validation	30	30	100
6. Improper socket	4	4	100
7. Using HTTP	222	222	100
8. Predictable seeds	0	0	—
9. Static salts	112	112	100
10. Electronic cookbook mode for symmetric cryptography	41	41	100
11. Static initialization vector	41	40	97.6
12. <1,000 PBE iterations	43	42	97.7
13. Broken symmetric cryptoalgorithm	86	86	100
14. Insecure public-key cryptography	12	12	100
15. Broken hash	138	138	100
Total	**1,153**	**1,135**	**98.4**

PBE: password-based encryption.

patterns. Table 2 presents multiple types of cryptographic API misuses in Java and the precision results of CryptoGuard analysis.[7] Precision is the percentage of true misuses out of all reported misuses. In complicated cases, we need to break down the detection plan into multiple steps, each mapped to a single round of static program analysis tasks. One such example appears in vulnerability type 14 in Table 2. In this case, detecting insecurely configured public-key cryptosystems requires two rounds of backward slicing, one round of forward slicing, and field sensitivity analysis. This manual mapping process is difficult and time consuming, requiring knowledge of both cryptographic coding and program analysis. Addressing the semantic gap is a critical process in building many security defenses, such as malware detection and network intrusion detection.

Hasn't This Kind of Detection Been Done Already?

Indeed, publications aiming at exposing insecure cryptographic coding exist, e.g., CryptoLint[8] and FixDroid.[9] These earlier efforts showed the existence of coding problems by reporting instances of insecure code discovered in real-world apps and software projects.

However, exposing vulnerabilities and designing industrial-strength scanners have entirely different requirements, goals, and testing metrics. For example, exposing vulnerabilities shows the existence of some misuses, while the latter (our work) strives to systematically produce reports that aim to cover a wide range of misuses. Thus, it is unclear how well their prototypes work in practice. To produce deployable-grade tools, accuracy and scalability challenges need to be addressed. The number of false positives must be low, without sacrificing the false-negative rate. Different goals create opportunities for different-but-equally valuable security policies, algorithms, insights, principles, and artifacts. Thus, we need both types of contributions as they are complementary.

We highlight one data flow analysis challenge associated with deployment-grade tool development work like ours, which an ad hoc vulnerability exposure study might not be able to address. CryptoGuard aims to capture data flows through fields (i.e., data members) of a class in Java. When an inappropriate value is passed through the field of an object, our analysis can accurately keep track of the field without involving other unnecessary fields of this object. Such an analysis is called *field sensitive* as it identifies data flows in the form of field access (e.g., getter methods). This data flow differs from the typical assignment- or argument-based data flows. If an analysis does not capture the influence through fields, then it is field insensitive. Field-insensitive analysis may cause false negatives, which means missing the detection of potentially vulnerable API misuses in our case.

Intuitively, field sensitive analysis—differentiating dataflows through different fields—is likely more accurate than field-insensitive analysis. However, there is a tradeoff. Maintaining field sensitivity may incur overhead. In particular, field-sensitive analysis may involve multiple levels of exploration, for example, as a result of multiple levels of getters, i.e., a getter within another getter function. Exploring such data flow rabbit holes takes time, yet the obtained results may or may not be relevant to security.

Thus, one needs to characterize how the depth-of-field-sensitivity exploration, that is, the number of getter

methods in a call stack that the analysis keeps track of, impacts accuracy and scalability. What is the right depth of exploration to use? How would the choice of exploration depth impact detection accuracy and runtime? We conducted an experiment with 30 Apache root subprojects and varied the exploration depth from one to 10. A depth of one is where a field-related method is invoked by the main slice. The experiment evaluated our ability to detect constant values that were relevant to the use of cryptography.

We obtained some interesting findings. Although the choice of exploration depth impacts detection accuracy, our results showed that a depth of one provides a good balance in our experiments.[7] How the depth-of-field exploration impacts detection accuracy (represented by F1 scores) is shown in Figure 1. In this experiment, regardless of the analysis depth, the number of crypto-relevant constants detected stays the same (the horizontal line in Figure 1), which we manually confirmed. The F1 score, based on both precision and recall, peaked at a depth of one. We observed that the deeper analysis sometimes results in more pseudoinfluencers, i.e., the discovered constants that do not have security impacts, making the F1 score drop. The runtime evaluation showed that increasing the analysis depth does not significantly increase runtime.[7] Based on these results, we set the field-sensitive analysis depth to one in all later experiments.

This kind of rigorous and intricate field-sensitive analysis improves accuracy and provides useful insights for understanding detection capabilities. In contrast, prototypes designed only to expose vulnerabilities often miss such insights.

Then, for a researcher who would like to contribute to deployment-grade code scanners, what is the number-one priority? What is the biggest feature that would make such scanners useful in practice?

A Deployment Enabler: False-Positive Reduction

Arguably, precision is the number one requirement of a code scanner in practice. All practical static program analyses are approximations; they estimate program behaviors. This estimation may cause errors such as overestimating vulnerabilities, which would generate false positives, i.e., false alarms. Confirming whether an alert is real or not needs to be done manually by experts and is often time consuming. This section explains how CryptoGuard minimizes false positives in Java API misuse detection.

Our approach can be understood as a conditioned program slicing. Once program slices are obtained, we scan them for suspicious elements such as hardcoded passwords. However, our initial attempt at detection returned many false positives, i.e., many irrelevant constants were flagged. Vulnerability types 1, 2, 3, 9, 11, and 12 (see Table 2) produced most of the false alarms because these misuses are constant oriented. For example, the detection of the first three vulnerability types searches for hardcoded keys and KeyStore passwords. Flagging all constants in a slice may create false positives as some constants may have nothing to do with security or are incompatible with the context (e.g., type mismatch). In other words, not all constants appearing in cryptographic APIs are important. Some of them do not impact security and can be safely ignored. We refer to these constants as *pseudoinfluencers*.

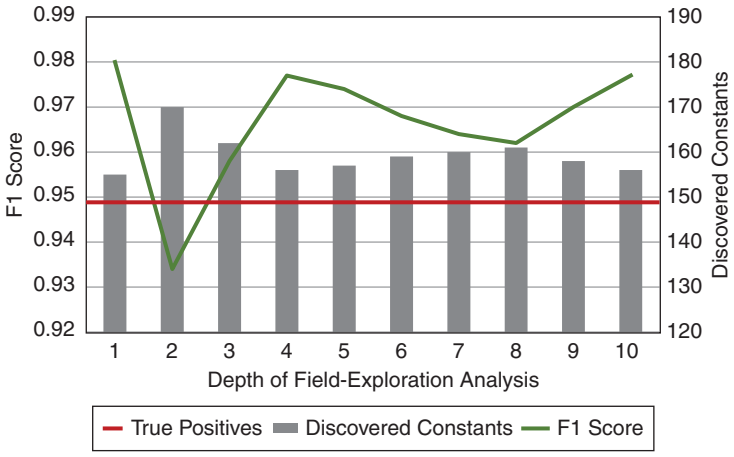

Figure 1. The impact of field-exploration depth (in the *x*-axis) on detection accuracy,[7] which is represented by the F1 scores (left *y*-axis), number of discovered constants (right *y*-axis), and number of crypto-relevant constants discovered (149, the horizontal line).

As a pseudoinfluencer example, an off-the-shelf detection method would report "UTF-8" as a hardcoded key when seeing the code

```
byte[] keyBytes = key.getBytes ("UTF-8").
```

However, `UTF-8` describes a character encoding and can be safely excluded. The sizes and indices of arrays are another type of pseudoinfluencer and can be removed from slices. We summarize these pseudoinfluencers into categories and identify them by our refinement insights.

Another category of pseudoinfluencer is identifiers, specifically those of value sources. For example, in `map.get("ENCRYPT_KEY")`, `"ENCRYPT_KEY"` is an identifier (like a name) used for retrieving a value from the Java Map data structure, not the actual element. Thus, this type of identifier can be safely ignored. Array indices are another example of constants that should be excluded as they do not cause cryptographic API misuses.

We developed five language-specific refinement algorithms to automatically reduce false positives. These five types of refinement are summarized by collecting and analyzing the sources that cause false positives. We find that most of them are derived from language-specific constants. All five language-specific strategies in CryptoGuard are for systematically removing irrelevant constants or predictable values from slices, including the removal of state indicators, resource identifiers, bookkeeping indices, contextually incompatible constants, and constants in infeasible paths.[7] In our experiments, the most effective refinement insight for Apache and Android code is the removal of array or collection of bookkeeping information. These five types of refinement algorithms drastically reduced the number of total alerts by 76% for 46 Apache projects and by 80% for 6,181 Android apps tested. Because Apache projects are open source, we were able to manually confirm that all the removed alerts are indeed false positives.

With refinements, CryptoGuard's accuracy is of deployment grade. Our manual source-code analysis on the 1,153 non-PRNG alerts confirmed 98.4% precision (Table 2), which would be reasonably manageable in a real-world deployment. A manual investigation reveals that some false positives are related to irrelevant constants discovered by the field analysis. A false-positive case is due to path insensitivity in our current data flow analysis. CryptoGuard cannot tell whether a PRNG method is used in a security sensitive context or an insensitive context, which also generates false positives. We could not evaluate the false-negative rate in this experiment due to the lack of ground truth. This problem motivated us to produce two benchmarks, which are described next.

Benchmarks and Comparison With the State of the Art

As there was not a benchmark at the time, all of our ground-truth confirmation was performed manually, which was a time-consuming and tedious task. This effort strongly motivated our later work on benchmark development, including the CryptoAPI-Bench, with 181 manually created test cases, and ApacheCryptoAPI-Bench, with 121 test cases from 10 real-world Apache software projects.[10]

For the first time, these benchmarks enable quantitative and side-by-side comparisons with other state-of-the-art solutions from industry and academia. Because security coverages of the four tools vary, our comparison needs to be conducted within the scope for which the tool is designed. In other words, we avoid testing a tool against test cases that the tool is not designed to detect. To address this requirement, our comparisons are carefully performed on the common subset of the vulnerability types, i.e., the shared security capabilities across all four tools being evaluated. Because of the lack of documentation in some tools, we need to experimentally identify the shared security capability. We infer a tool's coverage based on whether or not it ever generates any alert in that category.

For ApacheCryptoAPI-Bench, CryptoGuard reported the highest number of true positives, whereas the free online version of a commercial tool missed the most cases.[10] Interestingly for SpotBugs, although it performed miserably (0% recall and precision) under the 84 advanced test cases in CryptoAPI-Bench, it did quite well under ApacheCryptoAPI-Bench, successfully detecting many misuses. These results suggest that real-world code is likely much simpler than manually created advanced test cases in terms of data flow patterns. With ground truth, these benchmarks also enable researchers to investigate and compare false negatives. CryptoGuard's small number of false negatives come from either the truncated field-exploration analysis or vulnerabilities that are not yet covered. Although it performed well in the two benchmarks, we admit that it is possible for CryptoGuard to miss some cases neglected by the benchmark coverage.

Hardening Apache Projects

CryptoGuard also enabled us to identify and report several cryptographic API misuses in high-profile Apache projects, including various kinds of predictable secrets, vulnerable TLS/SSL certificate verification, hardcoded salts, and insufficient iteration counts in password-based encryption (PBE).[7] The Internet Engineering Task Force (IEFT) recommends a minimum iteration count of 1,000 for PBE, a popular encryption approach in real-world software projects. A widely used project for managing Hadoop security sets the iteration to the length of the password plus one, which is too small. The same method also used the password's MD5 hash as salt, which is insufficiently random. The vulnerable code is shown in "Listing 1." In addition, the iteration's dependence on the length of the password may also create a timing side-channel—an adversary capable of measuring PBE execution time (e.g., in multitenant environments) might be able to infer the length of the password. After our disclosure, several Apache projects responded to the vulnerabilities, including Ranger, Tomcat, Hadoop, Hive, Spark, Ofbiz, and Ambari. For example, Apache Spark removed the support of the dummy hostname verifier and dummy trust store. Apache Ranger fixed constant default values for PBE and insecure cryptographic primitives.[7]

Listing 1. A real-world PBE code snippet with multiple vulnerabilities. The salt is computed as the MD5 hash of the password (lines 2 and 3). As a result, an adversary may quickly recover the password from a leaked salt. In addition, the iteration count is set as the length of the password plus one, which is far less than the required 1,000 (line 6).

```
1  PBEKeySpec getPBEParameterSpec(String
   password) throws Throwable {
2     MessageDigest md = MessageDigest.
      getInstance(MD_ALGO);//MD5
3     byte[] saltGen = md.digest(password.
      getBytes());
4     byte[] salt = new byte[SALT_SIZE];
5     System.arraycopy(saltGen, 0, salt,
      0, SALT_SIZE);
6     int iteration = password.toChar
      Array().length + 1;
7     return new PBEKeySpec(password.
      toCharArray(), salt, iteration);}
```

Frustratingly, not all reported high-severity issues are fixed because some groups require an executable exploit demonstration, and our small team just does not have the resources to develop them. A possible research direction is to design an easy-to-use framework that could automate cryptographic exploit generation—sort of like "metasploit" for cryptographic code. In some other cases, developers acknowledged the issue,; however, the code could not be hardened due to backward compatibility or humanless deployment environments.[7]

For Android apps, 95% of API misuses came from libraries, including those from Google, Facebook, Apache, Umeng, and Tencent, not from the apps' own code. We also found that 25.3% of Android apps had a dummy trust manager (vulnerability type 5), higher than that of Apache projects (11.7%).

Oracle's Integration and Industrial Scanning Scalability

Oracle Labs Australia implemented CryptoGuard's detection approach in their internal checker Parfait,[11] with the help of Ya Xiao as a summer intern in 2019. Parfait[12] is a highly scalable static code checker designed for analyzing codebases in the scale of millions of lines.[11,13] Its development led by Cristina Cifuentes started in 2007 and spans considerably more than a decade. Parfait uses low level virtual machine not Soot, thus CryptoGuard's detection and refinement approach needed to be reimplemented for Parfait.

We conducted experiments to evaluate Parfait's new cryptographic code-scanning capability. When tested on 11 large codebases (nine Oracle internal products and two open source projects), Parfait's cryptographic scanner generated 42 alerts, all of which were manually verified as true positives (100% precision).[11] The low number of false alarms reported in this industrial setting experiment further confirms the deployability of our detection approach. Weak salts, low iteration counts in PBE, and bypassing certificate verification were among the issues identified. An evaluation of Parfait on CryptoAPI-Bench gave a similar accuracy performance as CryptoGuard's, as expected.

A static analysis with ultralow false positives is possible. The widely held belief that static analysis generates too many false alarms to be practically useful is not accurate. As demonstrated by CryptoGuard[7] and Oracle's Parfait,[11] there exist application-specific methods that systematically eliminate categories of false positives. In our problems, false positives are in the form of irrelevant constants. Identifying and systematizing the causes of false positives requires several months' one-time manual effort by Sazzadur Rahaman, yet the payoff is substantial.

Both CryptoGuard and Parfait spent substantial efforts on scalability, which is equally as important as accuracy for a deployment-quality code scanner. Understandably, developers prefer faster scanners that support continuous integration and delivery practice. Parfait has a creative, layered scheduling architecture that triages program analysis tasks from quickest to slowest, aiming at maximizing the number of tasks completed and issues found in a given period.[12] Backward slicing analyses are broken down and assigned to different layers based on their call depth. Specifically, at each layer, if the analysis is not conclusive (e.g., requiring the inspection of another method), then further analyses will be scheduled at the next layer. This layer-based scheduling approach is different from the conventional approach of fully completing one analysis before starting another. Layered scheduling allows short program analysis tasks to be executed first, conceptually similar to a breadth-first approach (as opposed to a depth-first search). In practice, complex analysis paths do not necessarily yield more vulnerabilities as CryptoGuard made similar observations in its field-sensitivity exploration experiment (see Figure 1). As illustrated in Figure 2, Parfait's layered scheduling design pays off; it finishes within 10 min on a typical workstation for most of the production codebases screened, with multiple projects having millions of lines of code.

With the focus on cryptographic code, CryptoGuard speeds up its screening by excluding irrelevant code and analyzing independent subprojects concurrently. In our case, building a complete general-purpose data flow graph is unnecessary as the cryptographic functionality

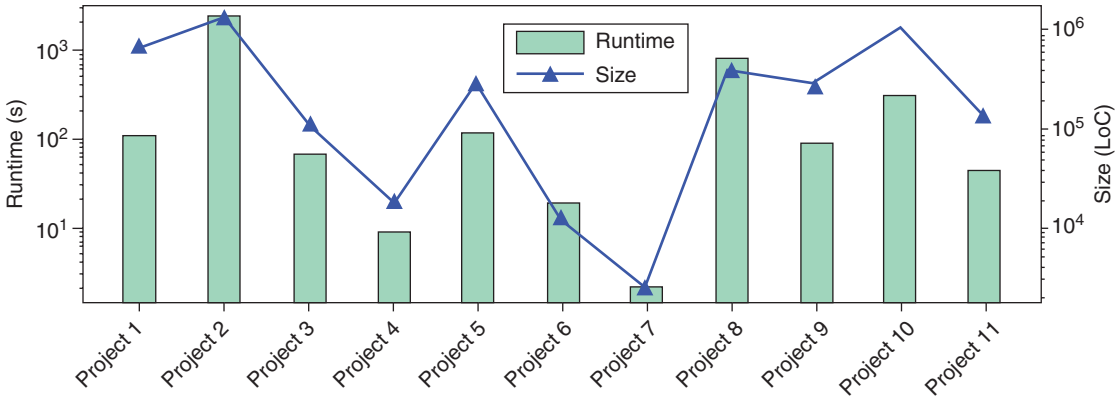

Figure 2. The runtime (left *y*-axis) of Parfait for screening 11 real-world codebases whose lines of code are also shown (right *y*-axis), both in log scale.[11]

is often confined within a small fraction of the project. Our data flow analysis is demand driven; it starts from the slicing criteria and propagates only to necessary elements. Thus, this specialized analysis avoids touching irrelevant subprojects and code regions. For Apache, CryptoGuard's average observed runtime was 3.3 min, with a median of approximately 1 min on a typical workstation. Including the cutoff ones, the average runtime and median were 3.2 and 2.85 min, respectively. (We terminated the unfinished app analysis after 10 min.) When evaluating 30 randomly selected Apache root subprojects [the lines of code (LoC) ranging from 471 to 1 K] and 30 Android applications (the LoC ranging from 1,453 to 0.4 K), CryptoGuard demonstrated higher efficiency than a leading academic solution in this space[14].

Exciting Future Directions

An exciting future direction is automatic code generation and suggestion, which aims at reducing or eliminating error-prone manual coding. The great success of natural language processing motivates the research of harnessing machine learning techniques to generate code. Programming languages have rigid structures, which are seemingly easier to predict than natural languages. However, they have a low tolerance for syntactical or semantic mistakes, especially for cryptographic code. Clearly, machine learning models for natural language generation cannot be directly applied to code. Large-scale code data sets and benchmarks also need to be developed for systematically measuring the accuracy of code-completion solutions. Such measurement efforts would be equally as important as proposing new neural network models.

Another direction is to create an easily configurable detector to automatically generate detection algorithms on the input of seminatural language statements specifying misuse patterns. Such a transformation capability would be powerful, making it easier to extend CryptoGuard by nonexperts.

For further reducing false alarms, there is a need to approximate and prioritize security-sensitive code regions. For example, raise an alert only when Random is used for security-sensitive tasks, such as generating session IDs and password reset links, and needs to be replaced by SecureRandom. One approach is to estimate the sensitivity of programming contexts.

We can leverage CryptoGuard's success with Java to cover other programming languages such as Python. We have also begun examining authentication and authorization misuses in the Java Spring enterprise framework.[15] The abuse of Spring security's customization capabilities is a source of application insecurity, such as lifelong access tokens, disabling CSRF protection, and hardcoded secrets.

Making tools like CryptoGuard fit into the larger toolchain used in industrial software development settings would help increase the adoption of security solutions. For example, we could produce development-time code-scanning capabilities in the form of compiler plug-ins and integrated development environment (IDE) plug-ins. We have demonstrated both Maven and Gradle compiler plug-ins for CryptoGuard. Ongoing work is on building IDE plug-ins, which will help detect and correct misuses early. One technical focus is to support the ability to analyze partial code. Another future direction is to systematically measure the usability of code-scanning tools in practice, in particular analyze the factors that contribute to developers' code change decisions, e.g., how to best provide information to help developers decide whether or not to modify the code region as suggested by tools.

Community support is crucial to promote practical deployment-quality tools. To broaden the

definition of novelty by recognizing deployable security will better align the most pressing cyberspace needs with researchers' efforts. Several notable initiatives have already started doing so, including the IEEE Secure Development Conference (sponsored by the IEEE Computer Society Technical Committee on Security and Privacy), the Annual Computer Security Applications Conference's (ACSAC)'s hard topic theme on deployable and impactful security, the *ACM Digital Threats: Research and Practice* journal, the Real World Crypto Symposium, and the Transition to Practice designation in the National Science Foundation funding programs. In addition, some conferences such as ACSAC and ACM Conference on Data and Application Security and Privacy encourage and incentivize artifacts and data submissions, which are all extremely encouraging.

Democratizing security knowledge is also a must. It will help more software developers and practitioners understand cutting-edge cybersecurity findings, well beyond our small group of researchers. It is important to consciously develop outreach resources, such as tutorials, short lessons, videos such as the "Introduction to Software Security" video series by Elisa Heymann and Barton Miller,[16] and security contests such as Build It, Break It, Fix It secure-coding competition series.[17] These types of efforts need support and recognition from our cybersecurity community. ■

Acknowledgments

Funding support for CryptoGuard includes U.S. Office of Naval Research grant number ONR-N00014-17-1-2498 and National Science Foundation grant number CNS-1929701.

References

1. N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, "Secure coding practices in Java: Challenges and vulnerabilities," in *Proc. 40th Int. Conf. Softw. Eng. (ICSE)*, Gothenburg, Sweden, May 2018, pp. 372–383, doi: 10.1145/3180155.3180201.
2. "Developer survey results." Stack Overflow. https://insights.stackoverflow.com/survey/2019 (Accessed: May 11, 2022).
3. F. Fischer *et al.*, "Stack overflow considered harmful? The impact of Copy&Paste on Android application security," in *Proc. 38th IEEE Symp. Security Privacy (S&P)*, San Jose, CA, USA, May 2017, pp. 121–136, doi: 10.1109/SP.2017.31.
4. "Tweet by _dvorak_." Twitter. https://twitter.com/_dvorak_/status/679109591708205056 (Accessed: May 11, 2022).
5. S. Checkoway *et al.*, "Where did I leave my keys?: Lessons from the Juniper Dual EC incident," *Commun. ACM*, vol. 61, no. 11, pp. 148–155, 2018, doi: 10.1145/3266291.
6. M. D. Weiser, "Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method," Ph.D. thesis, Univ. Michigan, Ann Arbor, MI, USA, 1979.
7. S. Rahaman *et al.*, "CryptoGuard: High precision detection of cryptographic vulnerabilities in massive-sized Java projects," in *Proc. 26th ACM Conf. Comput. Commun. Security (CCS)*, London, U.K., Nov. 2019, pp. 2455–2472, doi: 10.1145/3319535.3345659.
8. M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proc. 20th ACM SIGSAC Conf. Comput. Commun. Security (CCS)*, Berlin, Germany, Nov. 2013, pp. 73–84, doi: 10.1145/2508859.2516693.
9. D. C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl, "A stitch in time: Supporting Android developers in writing secure code," in *Proc. 24th ACM Conf. Comput. Commun. Security (CCS)*, Dallas, TX, USA, Nov. 2017, pp. 1065–1077, doi: 10.1145/3133956.3133977.
10. S. Afrose, Y. Xiao, S. Rahaman, B. P. Miller, and D. (Daphne)Yao, "Evaluation of static vulnerability detection tools with Java cryptographic API benchmarks," *IEEE Trans. Softw. Eng.*, early access, Feb. 2022, doi: 10.1109/TSE.2022.3154717.
11. Y. Xiao, Y. Zhao, N. Allen, N. Keynes, D. (Daphne) Yao, and C. Cifuentes, "Industrial experience of finding cryptographic vulnerabilities in large-scale codebases," *ACM Digit. Threats, Res. Pract.*, to be published, doi: 10.1145/3507682.
12. C. Cifuentes and B. Scholz, "Parfait: Designing a scalable bug checker," in *Proc. Workshop Static Analysis*, Tucson, AZ, USA, Jun. 2008, pp. 4–11, doi: 10.1145/1394504.1394505.
13. "Parfait project details." Oracle Labs. https://labs.oracle.com/pls/apex/f?p=94065:12:17236785846387:13 (Accessed: May 11, 2022).
14. S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An extensible approach to validating the correct usage of cryptographic APIs," in *Proc. Eur. Conf. Object-Oriented Program. (ECOOP)*, Amsterdam, The Netherlands, Jul. 2018, pp. 1–27, doi: 10.4230/LIPIcs.ECOOP.2018.10.
15. M. Islam, S. Rahaman, N. Meng, B. Hassanshahi, P. Krishnan, and D. (Daphne)Yao, "Coding practices and recommendations of spring security for enterprise applications," in *Proc. IEEE Secure Development (SecDev)*, May 2020, pp. 49–57, doi: 10.1109/SecDev45635.2020.00024.
16. E. Heymann, B. P. Miller, and L. Kohnfelder. (2022). Introduction to software security. [Online]. Available: https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/
17. A. Ruef, M. Hicks, J. Parker, D. Levin, M. L. Mazurek, and P. Mardziel, "Build it, break it, fix it: Contesting secure development," in *Proc. 23th ACM SIGSAC Conf. Comput.*

Commun. Security (CCS), Vienna, Austria, Oct. 2016, pp. 690–703, doi: 10.1145/2976749.2978382.

Danfeng (Daphne) Yao is a professor of computer science at Virginia Tech, Blacksburg, Virginia, 24061, USA. Her research interests include software and system security, with a special focus on measurable guarantees and performance. Yao received a Ph.D. in computer science from Brown University. She is an ACM Distinguished Scientist and a Senior Member of IEEE. Contact her at danfeng@vt.edu.

Sazzadur Rahaman is an assistant professor of computer science at the University of Arizona, Tucson, Arizona, 85721, USA. His research interests include building robust systems and methodologies for security. Sazzadur received a Ph.D. in computer science from Virginia Tech. Contact him at sazz@cs.arizona.edu.

Ya Xiao is a Ph.D. candidate with the Virginia Tech Department of Computer Science, Blacksburg, Virginia, 24061, USA. Her research interests include software security, deep learning, program analysis, and applied cryptography. Xiao received a master's from Beijing University of Posts and Telecommunications. Contact her at yax99@vt.edu.

Sharmin Afrose is a Ph.D. candidate with the Virginia Tech Department of Computer Science, Blacksburg, Virginia, 24061, USA. Her research interests include trustworthy machine learning in health care and software security. Afrose received a bachelor's in computer science from Bangladesh University of Engineering and Technology. Contact her at sharminafrose@vt.edu.

Miles Frantz is a Ph.D. candidate with the Virginia Tech Department of Computer Science, Blacksburg, Virginia, 24061, USA, where he received a master's in computer science. He research interests include creating code analysis tools to assist software developers. Contact him at frantzme@vt.edu.

Ke Tian is a senior machine learning engineer at Palo Alto Networks, Santa Clara, California, 95054, USA. His research interests include cybersecurity, anomaly detection, and applied machine learning. Tian received a Ph.D. in computer science from Virginia Tech. Contact him at ketian.yy@gmail.com.

Na Meng is an associate professor of computer science at Virginia Tech, Blacksburg, Virginia, 24061, USA. Her research interests include software engineering, programming languages, software security, and artificial intelligence. Meng received a Ph.D. in computer science from the University of Texas at Austin. Contact her at nm8247@vt.edu.

Cristina Cifuentes is a senior director of R&D at Oracle Labs Australia, Brisbane, Queensland, 4000, Australia, serving as director of the Software Assurance organization. Her research focuses on intelligent application security, aiming at making intelligent security of applications a reality, at scale. Cifuentes received a Ph.D. in computer science from the Queensland University of Technology. Contact her at cristina.cifuentes@oracle.com.

Yang Zhao is a research engineer at Oracle Labs Australia, and is currently a solicitor and software developer at IT and Startup Lawyers, Brisbane, Queensland, 4000, Australia. His research interests include software security, program analysis, model checking and machine learning. Zhao received a Ph.D. in computer science from the University of Wisconsin–Milwaukee. Contact him at yang.yz.zhao@oracle.com.

Nicholas Allen is a principal research engineer at Oracle Labs Australia, Brisbane, Queensland, 4000, Australia. His research interests include scalable software analysis. Allen received a bachelor's in software engineering from the University of Queensland. Contact him at nicholas.allen@oracle.com.

Nathan Keynes is the vice president of technology at Shorthand, Brisbane, Queensland, 4000, Australia. His research interests include static program analysis, systems integration, and e-commerce. Keynes received a bachelor's from the University of Queensland. Contact him at nkeynes@deadcoderemoval.net.

Barton P. Miller is a Vilas Distinguished Achievement Professor and Amar & Belinder Sohi Professor in Computer Sciences at the University of Wisconsin–Madison, Madison, Wisconsin, 53706, USA. His research interests include software security, tools for high-performance computing, binary program analysis, and instrumentation technologies. Miller received a Ph.D. in computer science from the University of California, Berkeley. He is a Fellow of the Association for Computing Machinery. Contact him at bart@cs.wisc.edu.

Elisa Heymann is an associate professor at the Autonomous University of Barcelona, Barcelona, 08193, Spain, and a senior scientist at the National Science Foundation Cybersecurity Center of Excellence

with the University of Wisconsin–Madison, Madison, Wisconsin, 53706, USA. Her research interests include software security and assurance. Heymann received a Ph.D. in computer science from the Autonomous University of Barcelona. Contact her at elisa@cs.wisc.edu.

Murat Kantarcioglu is the Ashbel Smith Professor of Computer Science at the University of Texas at Dallas, Dallas, Texas, 75080, USA. His research interests include working on security and privacy issues raised by data mining and machine learning applications, such as social networks, databases, and health care. Kantarcioglu received a Ph.D. in computer science from Purdue University. He is a distinguished member of the Association for Computing Machinery. He is a Fellow of IEEE and the American Association for the Advancement of Science. Contact him at muratk@utdallas.edu.

Fahad Shaon is the CTO of Data Security Technologies, Richardson, Texas, 75083, USA. His core research interests include secure cloud computing, trusted computing, and program analysis. Shaon received a Ph.D. in computer science from the University of Texas at Dallas. Contact him at fahad@datasectech.com.