Examining Cryptography and Randomness Failures in Open-Source Cellular Cores

K. Virgil English
kvenglis@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Nathaniel Bennett
bennet.n@ufl.edu
University of Florida
Gainesville, Florida, USA

Seaver Thorn
swthorn@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Kevin R.B. Butler
butler@ufl.edu
University of Florida
Gainesville, Florida, USA

William Enck
whenck@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Patrick Traynor
traynor@ufl.edu
University of Florida
Gainesville, Florida, USA

ABSTRACT

Industry is increasingly adopting private 5G networks to securely manage their wireless devices in retail, manufacturing, natural resources, and healthcare. As with most technology sectors, open-source software is well poised to form the foundation of deployments, whether it is deployed directly or as part of well-maintained proprietary offerings. This paper seeks to examine the use of cryptography and secure randomness in open-source cellular cores. We design a set of 13 CodeQL static program analysis rules for cores written in both C/C++ and Go and apply them to 7 open-source cellular cores implementing 4G and 5G functionality. We identify two significant security vulnerabilities, including predictable generation of TMSIs and improper verification of TLS certificates, with each vulnerability affecting multiple cores. In identifying these flaws, we hope to correct implementations to fix downstream deployments and derivative proprietary projects.

CCS CONCEPTS

• **Security and privacy** → *Cryptography*; **Mobile and wireless security**.

KEYWORDS

cellular core security, cryptography misuse, static analysis

ACM Reference Format:
K. Virgil English, Nathaniel Bennett, Seaver Thorn, Kevin R.B. Butler, William Enck, and Patrick Traynor. 2024. Examining Cryptography and Randomness Failures in Open-Source Cellular Cores. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY '24), June 19–21, 2024, Porto, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3626232.3653259

CODASPY '24, June 19–21, 2024, Porto, Portugal

ACM ISBN 979-8-4007-0421-5/24/06
https://doi.org/10.1145/3626232.3653259

1 INTRODUCTION

Cellular networks are the world's most important communication system. More people globally access the Internet from mobile devices than from laptop or desktop computers, and the ubiquity of cellular connectivity is a crucial reason why. The importance of cellular infrastructure is not limited to public networks; interest in deploying private 5G networks is high in sectors as varied as retail, manufacturing, natural resources, and healthcare. Assuring the security of cellular communication is a multi-faceted challenge requiring protection of information over the air from devices equipped with cellular hardware, to the radio access networks that receive the transmitted signal and forward it to the cellular core network. However, much of the previous research has considered over-the-air cryptographic ciphers [17, 18, 51, 54, 58, 60], while other aspects of cellular security have primarily been addressed in telecommunication standards.

Perhaps owing to the traditional opacity of cellular infrastructure, relatively little research has to date considered the security of cellular cores [11, 22], particularly with regards to information protection through cryptography. However, the rise of open-source cores provides an opportunity for insight into how information is secured in these implementations.

Industry commonly adopts open-source implementations, either directly as a solution or by wrapping core functionality into a well-maintained proprietary offering. Cellular is no exception. Private network deployments [1, 2] have begun increasing adoption of open-source cellular cores. Telecommunication provider interest is also evident through their sponsorship of several open-source celluar core projects [26, 27, 38, 43]. As with many technologies, the full extent to which open-source cellular cores propagate into real-world deployments may never fully be knowable. However, by identifying and fixing flaws in open-source cellular cores, these downstream projects and deployments will benefit.

This paper seeks to examine the use of cryptography and secure randomness in open-source cellular cores. Cryptography misuse has been extensively studied in other domains (e.g., Android [23, 24, 36, 55]). Use of insecure random number generators has also been studied for decades [14, 29, 34]. However, cellular systems are unique. First, unlike client software implementing custom protocols, cellular systems must implement 3GPP cellular standards in order to interoperate. However, they may still exist within legacy code and be exploited using downgrade attacks. Second, the 3GPP cellular

standards necessitate unique cryptographic values and primitives (e.g., RAND, CK, and RES). Based on our own investigation, we also found that open-source cellular cores commonly provide their own cryptography implementations. Therefore, existing library-specific detection patterns will not work. Third, these systems use secure randomness for celluar-specific functionality. For example, Temporary Mobile Subscriber Identities (TMSIs) provide privacy for users in the wireless portion of a cellular network.

In this paper, we create CellCrypt, which wraps CodeQL queries customized for cellular core implementations and processes the results. We perform a literature survey to define 13 rules to detect cryptography and secure randomness flaws in cellular core implementations. Our rules cover three categories: (1) insecure algorithms, (2) insecure use of randomness, and (3) insecure TLS configuration. We used our rules to analyze seven open-source cellular cores covering both LTE (4G) and 5G networks: free5GC [27], SD-Core [26], Open5GS [38], OpenAirInterface (OAI) CN5G [43], OAI EPC [44], NextEPC [33], and srsRAN [56]. After running CellCrypt on these cores, we manually reviewed results.

Our application of CellCrypt on the seven open-source cellular cores led to the discovery of two significant vulnerability findings. First, we found that the TMSIs generated in OAI-5G, OAI-LTE with Magma, and free5GC can lead to linkability attacks. We demonstrate an attacker can accurately link $TMSI_1$ of a subscriber to a newly generated $TMSI_2$ of the same subscriber, allowing an attacker to track UEs over time. Second, we find that certificate and hostname verification is disabled in OAI-LTE with Magma, OAI-5G, free5GC, and SD-Core even when the cores are configured by an operator to use TLS. This leaves communication between cellular core network functions (NFs) exposed to on-path attacks.

In addition to these key findings, CellCrypt identified other potential vulnerabilities. First, we found that SD-Core exposes an HTTP interface that signs OAuth tokens using a key encoded as a constant string. This code was removed from free5GC after the code was forked by SD-Core. Second, we found the OAI implementation pins a library dependency to a version with a known CVE that produces predictable IV values. OAI maintainers and derivative projects should use source composition analysis (SCA) tools to discover and remediate this vulnerability. Finally, we found that nearly all of the cores reference SHA1 and MD5 hash algorithms for backwards compatibility. We strongly encourage the 3GPP to deprecate MD5 and SHA1 as they are widly considered to be insecure.

In summary, we make the following contributions in this paper:

- *We define 13 misuse heuristics for cellular cryptography and secure randomness.* We survey prior work on cryptography misuse and define a set of misuse heuristics specific to cellular applications. Some traditional heuristics we adapt for the cellular domain, e.g., key randomization takes into account 5G NF communication. Other heuristics are cellular-specific, checking for AKA variables like RAND. We encode our rules in CodeQL to generically cover both Go and C/C++ cores.
- *We study seven cellular core implementations.* Until recently, nearly all cellular core implementations have been proprietary and difficult to obtain for study. This is one of the first papers to study the code in cellular core implementations.

Table 1: Cores Analyzed and Supported Cellular Generations

Cellular Core	Language	Version
Free5GC [27]	Go	5G
SD-Core [26]	Go	LTE, 5G
Open5GS [38]	C	LTE, 5G
OAI-5G [43]	C++	5G
OAI-LTE [44]	C++	LTE
NextEPC [33]	C	LTE
srsRAN [56]	C++	LTE

- *We identify two significant security vulnerabilities, each affecting multiple cores.* We found that OAI-5G, free5GC, and OAI-LTE with Magma generate preditable TMSI values and that OAI-LTE with Magma, OAI-5G, free5GC, and SD-Core do not properly verify TLS certificates when TLS is enabled.

The remainder of this paper proceeds as follows. Section 2 provides a brief history of cryptograpy flaws in cellular networks and motivates the need for dataflow analysis to accomplish our goal. Section 3 defines our rules. Section 4 describes our CodeQL rule encoding. Section 5 discusses results. Section 6 overviews related work. Section 7 concludes.

Responsible Disclosure: Vulnerabilities were found and confirmed in results shortly before submission. Responsible disclosure to affected parties is currently ongoing.

Availability: The source code for CellCrypt can be accessed at https://github.com/wspr-ncsu/cellcrypt.

2 BACKGROUND AND MOTIVATION

Cryptography and randomness vulnerabilities are not new topics for cellular systems. We begin with a brief history of flaws and motivate the need for dataflow analysis via a small experiment.

2.1 Vulnerabilities in Cellular Cryptography

Cellular technologies have struggled with cryptography vulnerabilities since 2G. Multiple cryptographic weaknesses in 2G GSM algorithms allowed attackers to easily break ciphers [50, 58]. COMP128, the cipher used for 2G GSM authentication and encryption, can be trivially broken [50, 58]. 3G replaced COMP128 with KASUMI and SNOW3G, and SNOW3G is still in use with certain 4G LTE and 5G algorithms. While 3G's cryptography is not as trivially exploitable as 2G, KASUMI has theoretical vulnerabilities, and MILENAGE side-channels on SIM cards allow for key extraction.

The 3G and 4G LTE Authentication and Key Agreement (AKA) protocols allow linkability attacks [18, 50]. In 4G, weaknesses in symmetric key mechanisms allow the RAND and AUTN values to be collected [18]. Previous generations also had known IMSI leakage problems, which caused the IMSI to be replaced by the SUCI in 5G [18, 50]. Research has also discovered theoretical weaknesses in the AKA [16, 21].

Problems with the implementations of cellular systems were the cause of several of the above vulnerabilites. Cryptographic side-channels from leaky MILENAGE and COMP128 implementations allowed key extraction [40]. Insecure manufacturer baseband implementations allowed for heap overflows in many devices [30]. Flaws in state machines resulted in acceptance of illegitimate base stations

Table 2: Number of results using dataflow analysis vs. grep

Cellular Core	R-5 Sinks	rand	.+(.*rand.*)	rand\h+\\[\\]byte	uint8_t\h+rand
free5GC [27]	3	427	97	1	–
SD-Core [26]	34	871	454	1	–
Open5GS [38]	17	228	53	–	32
OAI-5G [43]	37	147	241	–	93
OAI-LTE [44]	10	467	58	–	42
NextEPC [33]	17	57	31	–	24
srsRan [56]	7	169	119	–	77

for connection [46, 50]. However, these insecure implementation flaws are found on the *user equipment* connecting to the network, rather than the network software itself. Historically, the software in the cellular network has been proprietary with access to even binaries being extremely difficult to obtain.

2.2 Need for Dataflow Analysis

The design of LTE and 5G systems makes simple code checks difficult. In addition to having domain-specific algorithms and variables, many traditional algorithms are implemented directly rather than using well known libraries. Direct implementation can be attributed to reducing dependence on external libraries and patching the code for cellular compatibility. Unfortunately, the direct implementations often have custom names that complicate simple searches. Cellular-specific values such as `RAND`, `IK`, and `CK` also make searching for variables and function arguments difficult.

As discussed in Section 4, we use dataflow analysis to identify cryptography vulnerabilities in cellular core implementations. At a high level, rather than try to manually identify every potential function target, we look for matching function signatures and filter by their source library or header. We hypothesize that such an approach will allow us to identify dataflows in all cores without adapting the tool for each core. A question that stems from our hypothesis is whether it is useful in detecting misuse compared to manually searching the code.

We conducted a small experiment to demonstrate the need for dataflow analysis to detect cryptography misuse in our target cellular core implementations. As will be described in Sections 3 and 4, our Rule `R-5` identifies when `RAND` values are improperly randomized prior to use. Our experiment compares the number of results when using `grep` for several regular expressions to the number of unique dataflow sinks identified by our dataflow analysis. The number of sinks represents the number of code locations that a security analyst needs to investigate.

Table 2 shows the number results of our `grep` verses dataflow analysis experiment. The `rand` pattern represents a simple search for any variable, function, or other string that might be related to randomness. The `.+(.*rand.*)` and `uint8_t\h+rand` patterns represent matching Rule R-5 function signatures for sinks, with the latter pattern being more specific for C/C++ cores. The `rand\h+\\[\\]byte` pattern provides a similar pattern for Go cores. The table clearly shows the benefits of dataflow analysis over grepping for both the `rand` and `.+(.*rand.*)` patterns. The `rand\h+\\[\\]byte` pattern is too strict and misses many cases. The `uint8_t\h+rand` pattern results in a similar order of magnitude, but still produces significantly more results. Finally, in addition to reducing the number of code locations requiring inspection, dataflow analysis aids a security analyst with the corresponding sources and path information to drastically simplify manual confirmation.

3 RULES

The goal of this paper is to identify cryptography misuse in open-source cellular cores. Detection of cryptography misuse in software is a relatively-well studied topic, and therefore we conducted a literature review to identify a set of 13 rules appropriate for our target implementations. Specifically, we use six primary sources to derive our rules [12, 19, 23, 39, 47, 48].

Table 3 lists our 13 rules along with a brief description and associated analysis type. "A" or algorithm rules focus on the use of cryptography primitives. "R" or randomness rules check for insufficient randomness. "TLS" rules refer to TLS misconfigurations. We now describe the motivation for each rule. We discuss how we encode each rule into a program analysis check in Section 4.

3.1 Insecure Algorithms

We defined three rules that consider the use of insecure cryptographic algorithms. While the 3GPP specifications define which algorithms are allowed, we wanted to identify insecure algorithms used for nonstandard purposes or that may remain in legacy code.

A-1 (Insecure Symmetric Algorithms): *No insecure symmetric algorithms, as found by NIST or the community, may be used in the code base.* Insecure symmetric algorithms include DES, RC4, Cast5, and TEA. Target algorithms are derived both from literature [12, 19, 39, 47, 48] and NIST standards [15]. With NIST, we include algorithms that are being sunset.

A-2 (Insecure Hash Algorithms): *No insecure or collision-prone hash algorithms, as found by NIST or the community, may be used in the code base.* Examples of such algorithms are MD5 and SHA-1. These algorithms are derived from the same sources as A-1.

A-3 (Insecure Asymmetric Algorithm Use): *No insecure use of asymmetric algorithms.* In this context, insecure use is defined by block size. Sources define having less than a 2048 bit block size for traditional algorithms (e.g., RSA) or less than 224 bits for an elliptic curve as insecure [12, 15, 19, 39, 47, 48].

3.2 Insecure Use of Randomness

We defined six rules that consider the use of randomness within the cellular cores. Rule R-6 considers the source of randomness and how it is used for the dataflow sinks defined in Rules R-1 to R-5.

R-1 (IV Values are Unique): *All IV values are securely randomly generated and none are reused.* No constant IV values are used, and all generated IV values should use a cryptographically secure PRNG. Insecure IVs make it easier for attackers to predict future values. Well-known attacks have taken advantage of predictable IVs, such as the infamous BEAST attack [49]. Some literature sources only specify IVs must not be constant [23]. Others add they must be randomized [12, 19] and one qualifies the randomness as secure [39]. We defined our rule to meet the most stringent requirement.

Table 3: Analysis Rules Directing Dataflow and Code Analysis

ID	Name	Rule	Check Type
A-1	Insecure Symmetric Algorithms	No insecure symmetric algorithms, as found by NIST or the community, may be used in the code base	Code Syntax
A-2	Insecure Hash Algorithms	No insecure or collision-prone hash algorithms, as found by NIST or the community, may be used in the code base	Code Syntax
A-3	Insecure Asymmetric Algorithms	No insecure use of asymmetric algorithms	Code Syntax
R-1	IVs values are unique	All IV values are securely randomly generated and none are reused	Dataflow
R-2	Nonces are Random	All non-IV nonces are random and unique	Dataflow
R-3	Salts are unique	Salt values are random and unique when used	Dataflow
R-4	Keys are unique	All non-network keys are unique and when generated use a secure PRNG	Dataflow
R-5	Random Challenge Values Unique	RAND challenge values are unique and generated using a secure PRNG	Dataflow
R-6	Cryptographic Randomization	Insecure randomness is not used for situations requiring secure randomness	Dataflow
TLS-1	Secure TLS Cipher Suites	No insecure cipher suites are allowed by TLS	Code Syntax
TLS-2	Certificate Verification	Does not accept all certificates or otherwise disable certificate checks	Code Syntax
TLS-3	HostName Verification	Does not allow all hostnames or disable hostname verification	Code Syntax
TLS-4	Proper TLS Version	No TLS version below 1.2 is used	Both

R-2 (Nonces are Random): *All non-IV nonces are random and unique.* While IVs can be considered nonces, not all nonces are IVs. We have an additional rule for these nonce values. Many uses of nonce values do not require secure randomization. However, these values should still be pseudo-random and unique to prevent replay and other attacks. Our literature sources imply this rule by requiring no constant or predictable values [12, 19, 47].

R-3 (Salts are unique): *Salt values are random and unique when used.* Passwords are not common within a cellular core, and thus salt values are not commonly used. However, some of the open-source cores come with consoles or user front-ends for management. Passwords for these front-ends should use random and unique salts [12, 19, 39, 47, 48]. Note this rule does not check for proper storage of passwords or other password-based rules present in our reference literature. These challenges are out-of-scope for our work.

R-4 (Unique Keys): *All non-network keys are unique and when generated use a secure PRNG* As mentioned in Section 2, there is one primary key source in cellular AKA from which all other keys are derived. While key derivation occurs at all stages, key generation is not common within the core. Key randomization does not occur during derivation. Therefore this rule focuses on the use of hard-coded keys and the improper reuse of derived keys. Many sources also specify rules around key derivation and PBKDFv2 [12, 47, 48]. The KDF in the cores is unique to cellular. It does not require the same cryptographic checks as PBKDFv2 (e.g., iteration count) [6].

R-5 (Secure RAND Challenge Values): *RAND challenge values are unique and generated using a secure PRNG.* 3GPP TS 33.501 describes a single source of randomness used in coordination with a subscribers secret key to begin the AKA process [6]. This source of randomness, aptly named RAND, is the only source of cryptographic randomness in a fresh authentication. RAND should be randomly generated by the core and shared with the UE. The four values used for mutual authentication, RES*, XRES*, HRES*, and HXRES* are derived using this RAND. Additionally, all session keys are derived from RAND, K, and OP, with OP being a constant value [6]. The standards specify RAND must be generated using a PRNG and be unpredictable [4, 5].

R-6 (Cryptographic Randomness): *Insecure randomness is not used for situations requiring secure randomness.* Previous rules concerned the proper use of primitives, including both randomization and uniqueness. This rule focuses on the use of insecure randomness for cryptographic inputs. We define two categories off insecure randomness. The first is predictable seeding of a randomness source. An example would be seeding randomness with the current time. Second is the use of insecure randomness in any of our previous sinks. This encompasses flows directly from randomness sources to sinks, rather than using proper randomness as a barrier. We discuss this further in Section 4.3.

3.3 Insecure TLS Configuration

We defined four rules considering the configuration of TLS within the cellular cores. Developers historically misconfigure TLS in implementations, even when the default use is secure [24].

TLS-1 (Secure TLS Cipher Suites): *No insecure cipher suites are allowed by TLS.* This rule extends A-1 and A-2 to their acceptance in TLS communications. One of these ciphers may be allowed without being explicitly mentioned in the code. Therefore A-1 and A-2 would not catch the misuse. CryptoGo [39] and Ami et al. [12] explicitly mention insecure ciphers in TLS. The rest have rules regarding the insecure use of TLS and insecure algorithms, which can be taken together to derive this rule [19, 23, 47, 48].

TLS-2 (Certificate Verification): *TLS certificate verification is not disabled or set to accept all.* Certificate verification ensures the authenticity and integrity of a certificate. This process includes checking a certificate's expiration date, verifying the certificate chain to the root, and ensuring the certificate has not been revoked. If these checks are successful, then the certificate is valid for any host. Ami et al. [12], Rahman et al. [48], Piccolboni et al. [47], and Li et al. [39] all mention proper certificate verification. MITRE mentions certificate verification as CWE-295 [41], and OWASP [45] discusses multiple vulnerabilities linked to improper verification.

TLS-3 (Hostname Verification): *TLS certificates are checked to ensure a matching hostname.* Our literature sources and justification for TLS-3 are the same as TLS-2. Hostname verification checks extend certificate verification to whether the certificate matches the host to which the client is attempting to connect. Without hostname verification, a malicious attacker could present a valid certificate issued for a different host, leading to an on-path attack. In fact, MITRE has CWE-297 [42] under CWE-295 [41].

TLS-4 (Proper TLS Version): *Only TLS 1.2 or higher is accepted.* TLS-4 is derived from the 3GPP cellular standards, which explicitly state TLS 1.2 or higher must be used when HTTPS is used within the core [6].

4 CELLCRYPT

CellCrypt is an analysis tool built to study cryptography misuse in celluar cores. Specifically, CellCrypt wraps CodeQL queries customized for cellular core implementations and processes the results. We built CellCrypt on top of CodeQL, because it provides a declarative language for analysis across multiple programming languages, including both Go and C++, which are used by our target codebases. Figure 1 provides an overview of CellCrypt and our analysis methodology.

As shown in Figure 1, we considered seven open-source cores. CodeQL databases are extracted from this dataset and provided to CellCrypt (Section 4.1). A combination of 3GPP standards and extracted function metadata is used to encode the rules defined in Section 3. These encoded rules are provided to CellCrypt as custom CodeQL queries (Sections 4.2 and 4.3). Finally, CellCrypt uses a combination of automated processing and a manual filtering methodology to refine the results (Section 4.4).

4.1 Database Extraction

CodeQL performs its analysis on extracted databases. These databases contain all program information including the control flow graph (CFG), dataflow graph (DFG), and abstract syntax tree (AST). To create a database, CodeQL must observe the compilation process (for compiled languages) or execution of source code (for interpreted languages). All of the cores in our dataset are written in compiled languages. Therefore, CodeQL compiles the cores to extract the databases. CellCrypt uses Docker to perform the CodeQL compilation. A CodeQL Docker container wraps the core and executes it's build process. Build errors must be manually corrected.

Go-based cores require additional preparation prior to building, as indicated with the blue icons in Figure 1. CodeQL does not examine imported package source code when analyzing a Go program. Therefore data and control flow is not extended through these packages unless CodeQL is explicitly configured with mapping rules. As each package requires a mapping class, the volume of imported packages in our cores makes this approach infeasible.

Instead, we wrote the `CorePrep` script that automatically includes the code for all dependencies in the analyzed source code. `CorePrep` begins by performing a local download of imported packages using the `go mod vendor` command and subsequently removes the `go.mod` and `go.sum` files. `CorePrep` then goes through each Go file and uses `perl` regex to change all external imports to local imports. Once all imports are changed, the `go.mod` and `go.sum` files are regenerated. A CodeQL database extracted from prepared cores now includes source code and flows for the imported libraries.

4.2 Sources, Sinks, and Barriers

CodeQL supports both inter- and intra-procedural dataflow analysis and taint tracking. We use a combination of these flows for our analysis. Every CodeQL dataflow query has the three traditional node definitions: sources, sinks, and barriers. Sources and sinks mark the introduction and target points for the analysis. Barriers are dataflow nodes which flow must not move through to be positive. CodeQL uses these barriers to identify flows to exclude from results.

Sources: Dataflow and taint source definitions for all relevant rules across both languages follow the same logic. CellCrypt defines sources as an assignment to any variable of the same type as the corresponding sink. For instance, for a sink type `ByteArray`, the corresponding source is usually an assignment to a variable with type `ByteArray`. Exceptions are discussed with their relevant rules in Section 4.3. We identified instances where a variable of a different type (e.g., a string) is populated with a cryptographic value prior to being assigned to an already defined source. Randomization or constant value assignment can occur prior to these re-assignments. To account for such instances, CellCrypt finds the initialization of any variable with intra-procedural flow to a defined source.

```
1 int ogs_aes_setup_enc(uint32_t *rk, const uint8_t *key, int
      keybits);
2 int ogs_aes_setup_dec(uint32_t *rk, const uint8_t *key, int
      keybits);
3
4 void ogs_aes_encrypt(const uint32_t *rk, int nrounds,
5        const uint8_t plaintext[16], uint8_t ciphertext[16]);
6 void ogs_aes_decrypt(const uint32_t *rk, int nrounds,
7        const uint8_t ciphertext[16], uint8_t plaintext[16]);
```

Listing 1: Signatures for Custom AES in Open5GS

Sinks: CellCrypt dataflow and taint analyses generally use API calls to cryptography functions as sinks. Cryptography implementations vary greatly between the cores, from the language base library to completely custom implementations. For example, Open5GS has its own cryptography implementation with unique function signatures. Listing 1 shows function signatures for AES in Open5GS. The cores also use a range of external libraries.

For the cores written in Go (i.e., free5GC and SD-Core), CellCrypt uses the built-in and common base libraries to identify most cryptography sinks. Cellular-specific algorithms are not present in the base libraries, so are identified from the code base. We first went through the Go library documentation, marking specific arguments to relevant API calls as sinks. We then used CodeQL to extract all function signatures, their call locations, and their library from the cellular core implementations. From this metadata we marked cellular-specific sinks not present in the base library. The marked functions and arguments are encoded in CodeQL as sink classes.

For cores written in C/C++, sink identification was significantly more difficult. Instead of marking sinks by function, CellCrypt filters functions by headers. After extracting the same metadata as with Go, we used CodeQL to extract file information for all imported headers. We then examined and cross-referenced headers with function metadata to determine cryptography headers. A header was marked as a cryptography header if it contains at least one function signature appearing to perform a cryptography operation. We marked all functions in these header files as a sink. While this broader definition of a sink results in a larger number of results, it is less time consuming to filter the results than manually annotate every function in the core implementation.

Finally, we define our CodeQL rules to mark *arguments* to calls as sinks rather than function parameters. Doing so causes the CodeQL results to identify a taint sink each time a cryptography function is called, rather than one taint sink for each cryptography function. As such, the count of unique sinks and source/sink pairs are useful metrics for our analysis (see Section 5).

Barriers: CodeQL uses barriers to exclude "good" flows from the results. These barriers change based on rule (see Section 4.3). Our

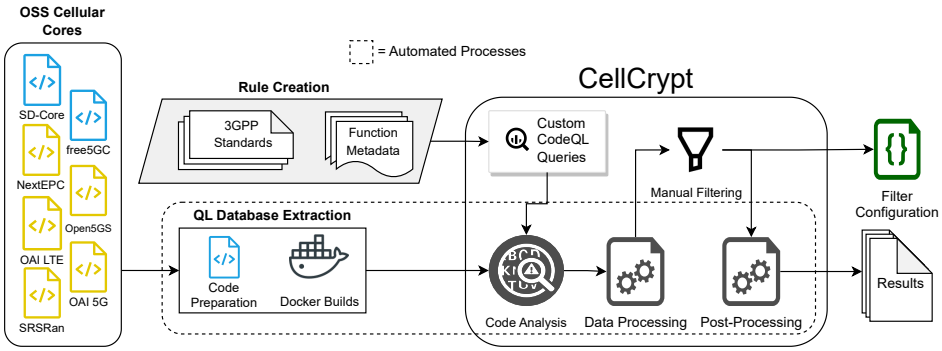

Figure 1: Overview of Analysis Methodology Using CellCrypt

two primary types of barriers are (a) value reassignment and (b) secure randomization. Value reassignment barriers are any assignment statements not preserving the previous data. Sources of secure randomization are identified during our sink identification process. A call to one of these functions is considered a barrier.

4.3 Encoding Rules

CellCrypt uses two types of rules: code syntax checks and dataflow queries. The type of each rule is indicated in Table 3. Rules are encoded for two languages, Go and C++. Each rule is encoded as a set of CodeQL queries. Encoding logic remains similar between languages and check types. For the dataflow analysis, the only difference between the cores is the sources, sinks, barriers for dataflow analysis. The TLS rules also follow different logic between the two languages. For each rule we give a description of it's encoding and any specifics in Go and C/C++.

4.3.1 Insecure Algorithms. Rules A-1 to A3 are encoded as static checks, which are discussed next.

A-1 (Insecure Symmetric Algorithms): A-1 checks for outdated or insecure symmetric key algorithms, and is encoded as a static check. We compiled a list of outdated or insecure algorithms from NIST standards [15]. From this list, we selected algorithms based on temporal and practical traits. In other words, based on recentness of the algorithm and how likely they are to be used. Likelihood of use is based on their incorporation into the Go base library. Specifically, we look for the following algorithms: XTEA, Cast5, DES, RC4, Blowfish, and TEA. Go separates cryptographic libraries by algorithm. For the Go code bases, CellCrypt looks for any imports of the identified libraries. With C/C++ code bases, CellCrypt checks cryptography headers for references to insecure algorithms.

A-2 (Insecure Hash Algorithms): Hash algorithms with known collisions violate A-2. To select which algorithms to encode for, we again checked NIST standards [15]. We select SHA1, MD5, MD4, and RIPEMD to be targets for A-2. For A-2 CellCrypt follows the same check logic as A-1.

A-3 (Insecure Asymmetric Algorithms): A-3 is dependent on block size rather than cipher. CellCrypt attempts to resolve the value of arguments in asymmetric API calls. Go only specifies block size in a call to the `GenerateKey()` method. CellCrypt checks for any calls to `GenerateKey()` with a value less than 256 (i.e., 2048/8, as values are in bytes not bits). CellCrypt also checks for the import of `crypto/dsa`, as DSA is an outdated algorithm. Go does not have built-in support for elliptic curve lower than 233 bits.

For C/C++ CellCrypt checks for function signatures indicating an `int` parameter and a byte array parameter. Byte array parameters can take the form of both `byte` and `uint8_t` arrays. Functions from a cryptography header are marked as sinks. CellCrypt checks identified `int` arguments for values less than 256 (i.e., 2048/8).

4.3.2 Insecure Algorithms. Encodings for Rules R-1 to R-6 (i.e., the randommness rules) are encoded as dataflow analyses.

R-1 (IV Values are Unique): Starting with R-1, both encodings for Go and encodings for C/C++ begin to follow a pattern. For Go, we identified the third argument of `NewCTR`, `NewCFBEncrypter`, `NewCBCEncrypter`, and `NewOFB` as sinks. C/C++ IV sinks are determined by function signature and header. CellCrypt marks any calls to functions originating from a cryptography header with an IV parameter as sinks. Sources for both Go and C/C++ follow the logic discussed in Section 4.2. CellCrypt uses two definitions of barriers, (1) calls to CSPRNG functions and (2) non-constant value reassignments.

R-2 (Nonces are Random): For R-2, we identified the `Seal()` function as a sink in Go. `Seal()` encrypts and signs a passed buffer. C/C++ sinks are identified in the same manner as R-1, but with a nonce function signature. Sources follow the logic in Section 4.2. Unlike the other rules, R-2 only has assignments as barriers. Nonces require uniqueness but not secure randomness.

R-3 (Salts are unique): Both R-4 and R-5 follow the same logic as R-1 for Go and C/C++. CellCrypt uses function calls with a salt function signature as sinks. For Go these primarily were password or key derivation functions, such as `Key` in `crypto/pbkdf2`. In C++ these were filtered by header, as with previous encodings.

R-4 (Keys are unique): Key randomization is difficult to check in the context of a cellular core. The key is pulled from the subscriber database. Further keys are derived from this key, which are sent between microservices. CellCrypt can not just check for flows without randomness, as in R-1, R-4, and R-5. All key flows would be marked positive.

5G uses a RESTful microservice structure in the core. Service endpoints are identified in the 3GPP standards. Some cores implement endpoints directly from the standards, usually through generation. Other cores contain custom implementations. CellCrypt automatically identifies these endpoints and excludes flows containing them. For Go, CellCrypt extracts endpoints from `Route`

objects. CellCrypt follows event listener state machines to identify endpoints. CellCrypt performs the same data flow logic as R-1 for both Go and C/C++.

R-5 (Random Challenge Values Unique): As mentioned in Section 2.1, 3GPP standards specify that `RAND` must be generated using a PRNG and be unpredictable [4, 5]. As `RAND` is a cellular-specific value used in cellular specific algorithms, we encode this rule with explicit sinks. Specifically, we locate the key and challenge derivation functions in each core and encode them as sinks. Barriers are secure PRNGs and value reassignments. `RAND` is also sent to the UE. Similar to R-3 we exclude network endpoints from our sources.

R-6 (Cryptographic Randomness): R-6 has two distinct encodings. One encoding takes bad randomness sources and finds dataflow to previously defined sinks. Examples of bad randomness are `rand()` in libc and `mpz_urandomb()` in GNU MP. Both `rand()` and `mpz_urandomb()` are known to be predictable [20, 28]. CellCrypt identifies flows from these functions to sinks for rules R-1 to R-5. A static check for bad PRNG seeds is the second encoding. CellCrypt identifies explicit bad seeds such as `srand(time(NULL))`, first by checking for calls to `time()`, then known seed functions for default values.

4.3.3 Insecure TLS Configuration. TLS rules (i.e., TLS-1 to TLS-4) are encoded as a combination of static checks and dataflow analyses.

TLS-1 (Secure TLS Cipher Suites): TLS in Go uses a `Configuration` struct. CellCrypt locates `Configuration` structs and checks for values that violate TLS rules. Improper values are then used as sources with flow to functions from `crypto/tls`. Improper values for TLS-1 are cipher flags containing insecure algorithms (e.g., TLS_RSA_WITH_RC4_128_SHA). C++ TLS libraries commonly use flags for configuration. CellCrypt checks for flags violating TLS-1.

TLS-2 and TLS-3 (Certificate / HostName Verification): In Go, hostname and certificate verification are disabled if the `InsecureSkipVerify` boolean is set to true. In line with TLS-1, CellCrypt uses `InsecureSkipVerify` fields set to true as sources. Sinks are then any operations from the `crypto/tls` library.

TLS-2 and TLS-3 for C++ is difficult. OpenSSL, GnuTLS, and Curl are the common TLS libraries in our dataset. These libraries use set flags to configure hostname and certificate verification. OpenSSL flags include `SSL_VERIFY_NONE`, which disables checks. GnuTLS flags include `GNUTLS_VERIFY_DISABLE_CA_SIGN` along with `GNUTLS_VERIFY_DISABLE_CRL_CHECKS`, and GnuTLS specific version flags. As 5G NFs are both clients and servers, `CURL_OPTS` flags are identified as well. Specifically `CURLOPT_SSL_VERIFYPEER` and `CURLOPT_SSL_VERIFYHOST` flags set with a false (0) value. We performed a simple code search to locate instances of these flags.

TLS-4 (Proper TLS Version): CellCrypt checks the `MinVersion` field in Go TLS configuration objects for a value other than `VersionTLS12` or `VersionTLS13`. Violating values are set as sources for data flow to `crypto/tls` sinks.

OpenSSL, GNU MP, and Curl version configurations are searched for in C++ cores. TLS-4 in C++ is our only rule where a true positive is a negative result. To achieve TLS-4, OpenSSL and CURL must *disable* prior versions. Thus, CellCrypt checks for this disable procedure and provide a result stating "Invalid TLS versions disabled" if all are detected. CellCrypt checks for calls to `SSL_CTX_set_min_proto_version` with `TLS1_2_VERSION`), which disables TLS1.0 and TLS1.1 in OpenSSL. CellCrypt also checks for calls to all OpenSSL flags from prior versions required to disable the proper TLS versions. These flags are `SSL_OP_NO_SSL*`, `SSL_OP_NO_TLSv1`, and `SSL_OP_NO_SSLv1_1`. GNU MP has multiple methods of setting version. The suggested method uses configuration fields disabling specific versions. We checked configuration files of positive results to determine if (GNUTLS) flags for TLS version are set.

4.4 Refining Results

CellCrypt processes analysis output automatically. However, due to the diverse nature of our code bases and limitations of CodeQL, we needed to perform some manual filtering of the results for bad sources and sinks. This filtering is core-specific, but filtering for each core followed the same rigorous algorithmic process. At a high level, the process proceeded as follows: (1) automated result processing, (2) filter bad sources, (3) filter bad sinks, (4) remove same-file flows, (5) examine remaining source/sink pairs.

Note the filtering process results in a reusable configuration. Subsequent runs of CellCrypt do not require manual filtering unless a core is added. This allows for modification of encodings without performing the entire filtering process again. Each filtering step is described below.

Step 1: Automated Result Processing: First, CellCrypt compiles results into JSON format and organizes them by rule per core. CellCrypt converts the raw output to JSON format with CodeQL utilities. Next, CellCrypt organizes the JSON outputs by core. Raw CodeQL output is a single large JSON object of source/sink pairs and their locations. CellCrypt determines the rule type (static check or data flow). Results are then organized by unique source, unique sink, and unique source/sink tuples for each rule. Counts for each, as well as total raw results, are also collected. Duplicates are removed by maintaining a list of seen source/sink pairs. Location information is the primary heuristic for this processing. Finally, CellCrypt places the raw results, processed results, and counts into a dataclass and converts to a JSON object. These JSON objects are compiled into a single file and output as processed results.

Step 2: Manually filter bad sources: Each processed result has a list of individual source and sinks. Some sources and sinks were incorrectly identified. We implemented a regex filter and manually go through each result. External libraries, network sources, and non-crypto sources are removed. Granularity of the filter is based on cross-referencing results. Generic filters, such as `metrics.c`, are identified in multiple cores with manual search. Network sources are filtered at a relative path granularity. Test files are also filtered with a broad `r'/tests{0,1}.*'` filter. This is applied to both sources and sinks. Results indicating cryptography (e.g., `oai-hss/src/hss_rel14/hsssec/src/kdf.c`) are assumed correct without examination.

Step 3: Manually filter bad sinks: Filters for test files and external libraries are added. Next, each sink is examined for relevance to cryptography. Sinks are filtered on a relative path granularity. Files indicating network operations are examined first. Hashmap and string operations are located next. Results indicating cryptography are assumed correct in-line with the source filtering process.

Step 4: Manually filter same-file flows: All remaining source/sink tuples are examined for same-file flows. If five or more tuples are in the same file and source/sink tuples are different pairings of the

Table 4: CellCrypt findings across Open-Source Cellular Cores–✗ indicates Vulnerabilities

Finding	Rule	free5GC	SD-Core	Open5GS	OAI-5G	OAI-LTE	NextEPC	srsRAN
Predictable Randomness	R-6	✗	✗	–	✗	✗	–	✗
Disabled Certificate Validation	TLS-2, TLS-3	✗	✗	–	✗	✗	–	–
Constant Unused Signing Key	R-4	Fixed in v3.2.1	✗	–	–	–	–	–
Deprecated Algorithms	A-2	SHA1, MD5	–	SHA1, MD5	SHA1, MD5	SHA1, MD5	SHA1, MD5	MD5
CVE-2022-46397	R-1	–	–	–	✗	–	–	–

same locations, the file is added to the same-file filter. While the cores all implement cryptography differently, common file names are found across cores. If same-file flow is indicated in a common file name in 3 or more cores, a file stem granularity filter is added. Otherwise filters are relative paths.

Step 5: Examine remaining source/sink pairs: Results for the core are reprocessed by Step 1 with the filters applied. Remaining results are examined by source/sink tuple. The source is checked for relevancy and missed randomization. Sinks are checked for relevancy. If both are found correct, the dataflow is followed manually to confirm the result.

5 RESULTS

Table 4 shows all findings uncovered by CellCrypt. We identify two key findings that are common across multiple independent cores, suggesting systematic issues in correctly implementing certain security requirements:

- Generated TMSIs in OAI-5G, free5GC and OAI-LTE with Magma are predictable which can enable linkability attacks. We demonstrate an attacker can accurately link a subscriber's $TMSI_1$ to a newly generated $TMSI_2$ of the same subscriber, allowing an attacker to track UEs over time.
- Certificate and hostname verification cannot be enabled in OAI LTE/5G, free5GC, and SD-Core even when the cores are configured by an operator to mandate TLS. This leaves communication between NFs exposed to on-path attacks.

These findings and their impact is discussed in Section 5.3. CellCrypt also finds several implementation-specific vulnerabilities highlighted in Section 5.5.

5.1 Dataset

Our analysis covers seven open-source implementations of cellular cores (1) Free5GC v3.0.2 (R15); (2) SD-Core v1.3.0 (R15); (3) Open5GS v2.5.6 (R15); (4) OpenAirInterface CN5G v1.4.0 (R14), (5) OpenAirInterface EPC v1.2.0 (R16) using Magma's MME; (6) NextEPC v1.0.1; and (7) srsRAN v23.04.1. OAI-LTE uses Magma's MME, so results for both are included in OAI-LTE with Magma. Most of these implementations are written in C/C++. Free5GC and SD-Core are written in Go. Each code base presents unique cryptographic considerations. For example, Open5GS implements cryptographic algorithms directly while others rely on standard libraries.

5.2 Data Characteristics

CellCrypt analyzes all projects with our encoded rules in CodeQL described in Section 4.3. We ran our analysis in a VM allocated 24 logical processors of an AMD EPYC 7302P 16-core processor and 128GB of memory, running Ubuntu 22.04. For a fresh run, CellCrypt averaged 0.6 hours and used 71GiB of RAM to analyze 8 million lines of code in C++ and Go across 7 code bases. Subsequent runs averaged 0.25 hours and 41GiB of RAM. Performance increase is a result of CodeQL caching behavior.

Table 5 shows result characteristics. The table compares raw CodeQL output (R) to final results after our filtering process discussed in Section 4.4. As indicated, all R-* rules are provided as counts of unqiue source/sink tuples. For Go cores (i.e., free5GC and SD-Core) this count is the same as total results. Go is an SSA language, so each flow has its own variables. In C/C++ cores source/sink tuples reduce redundant findings. CodeQL, as a query language, returns all result combinations which match the provided filters. This can result in multiple flows between the same source/sink pairs, e.g., around a boolean `if` statement.

Our filters achieved an 86.39% average refinement from raw output. R-4, keys are unique, showed the most overall reduction. We can attribute this to two factors. First, cryptographic functions outside of our rule set were identified e.g., CMAC. Second, network activity from the UE and other non-NF sources had to be filtered as well as outgoing network operations.

The next largest reduction occurred with R-1. Same-file flows between utility functions within code bases contributed to the disparity. For instance, 186 same-file flows in the Open5GS Zuc implementation were filtered. The least reduction occurs in TLS rules. This is expected due to their static check nature. For data flow, our cellular-specific rule R-5 fared the best. This is also expected, as cellular-specific variables are unlikely to be used elsewhere.

TLS-4 is shown as a pass/fail result. As mentioned in Section 4.3 TLS versions must be explicitly disabled. A true result indicates the presence of correct disabling flags. Results include checking of configuration files after `GNUTLS` is detected.

5.3 TMSI Linkability Attack

The goal of a Temporary Mobile Subscriber Identities (TMSIs) linkability attack is to associate several distinct TMSI identities over time to one particular subscriber, thereby nullifying the intended privacy guarantees offered by the TMSI. A TMSI is a temporary identifier of a subscriber assigned by the network to support identity concealment. TMSIs are required to be unpredictable by the 3GPP. Using cryptographically insecure randomness or predictable seeds allows attackers to predict the TMSI. The ability to predict TMSI assignments is sufficient information for widespread, long-term linkability attacks against all subscribers served by an MME/AMF[1], contingent on the TMSI reallocation strategy of the implementation. First, we describe the CellCrypt findings and how an attacker can

[1]An MME/AMF is responsible for managing mobile devices over large geographic areas (i.e., thousands of square miles)

Table 5: (R)aw and (F)iltered CellCrypt Results

	free5GC		SD-Core		Open5GS		OAI-5G		OAI-LTE		NextEPC		srsRAN	
	R	F	R	F	R	F	R	F	R	F	R	F	R	F
A-1	0	0	0	0	0	0	4	0	4	0	0	0	1	0
A-2	4	4	0	0	9	0	0	0	0	0	0	0	1	1
A-3	0	0	0	0	66	40	9	4	57	25	10	9	31	30
R-1*	1586	106	918	87	140	2	152	103	642	14	124	1	108	5
R-2*	289	16	35	16	105	42	26	6	1014	9	39	4	66	30
R-3*	0	0	80	0	0	0	0	0	822	0	0	0	0	0
R-4*	2350	23	1017	161	2351	33	335	8	6363	19	1165	17	221	113
R-5*	4	3	4	2	55	27	201	14	63	63	39	4	48	22
R-6*	6	6	0	0	0	0	6	6	3	3	0	0	5	3
TLS-1	0	0	0	0	5	0	0	0	4	0	5	0	0	0
TLS-2	36	36	43	43	1	1	12	12	1	1	0	0	0	0
TLS-3	36	36	43	43	1	1	12	12	1	1	0	0	0	0
TLS-4†	×		×		–		×		×		–		–	
Total	4297	226	2140	352	2733	146	757	165	8974	135	1382	35	481	204
Reduction	94.74%		83.55%		94.66%		78.20%		98.50%		97.47%		57.59%	

* Reported by Source/Sink Pairs; † TLS-4 is pass/fail, × indicates failure

use predictable randomness to predict TMSIs. Then, we describe the steps an attacker would take to achieve a full end-to-end attack.

Predictable Seeds for PRNG Algorithms: OAI-5G, SD-Core, Free5GC, and OAI-LTE with Magma (specifically the Magma MME) use predictable seeds for PRNG algorithms. We identify multiple sources of insecurely seeded PRNG across code bases. These insecure sources are used to generate the `RAND` field and to generate TMSIs. These predictable seeds are either timestamps or constant values; Table 6 shows how the PRNGs are seeded for each core. TMSIs are designed to be short-lived, such that actions or locations visited by a subscriber cannot be associated with that particular subscriber. The 3GPP standards for LTE and 5G state that TMSIs should be unpredictable and random [7].

To accurately predict future TMSIs, an attacker must determine when the cellular core's MME/AMF component was initialized. The attacker would first observe current TMSIs by passively sniffing cellular radio traffic. The attacker would then brute-force the time value within a reasonable window to see what initial seed would produce the observed TMSI values.

Cryptographically-Insecure PRNG Algorithms: OAI-5G, free-5GC, SD-Core, srsRAN, and Magma (in OAI-LTE) use cryptographically insecure PRNG algorithms for cryptography and privacy-sensitive operations. This vulnerability can lead to predictable TMSI and `RAND` values. Three PRNG algorithms are used across the four affected cores: Mersenne Twister, glibc `rand()` (which uses Linear Feedback Shift Register), and the Go `math.rand` function (which uses a Lagged Fibonacci Generator) [14, 59]. Each of these algorithms is known to be cryptographically insecure. The recovery of a sufficient number of outputs (ranging from 30 to 620, algorithm depending) will enable an attacker to construct the internal state of the PRNG and predict all future outputs. As a consequence, TMSIs can be predicted accurately.

Additionally, OAI-5G, free5GC, and srsRAN do not use PRNG algorithms to generate TMSIs. Instead, TMSIs are assigned using an incremented count, i.e., the TMSIs assigned are monotonically increasing. This approach is the easiest for an attacker to predict. If an attacker can connect a legitimate UE to the network and receive a TMSI, that attacker can simply increment upwards from that TMSI value to predict future values for other UEs.

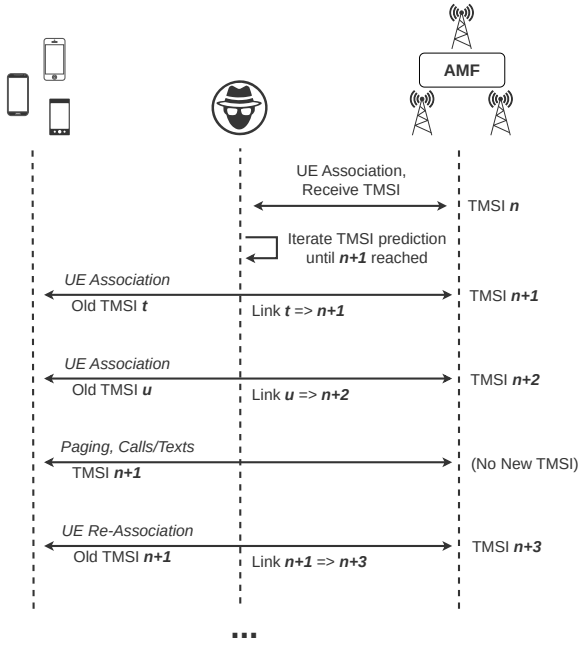

Figure 2: A wide-scale linkability attack made possible by predictable TMSI allocation. 5G network shown; also applicable to LTE.

Attack Description: To carry out a linkability attack, the attacker must be able to deterministically map each old TMSI value of a subscriber to the new TMSI value assigned whenever that subscriber is provided a new TMSI by the network. Figure 2 shows the attack.

Table 6: Patterns in TMSI Generation & Reallocation

Cellular Core	TMSI Generation Method	Randomness Seed	Messages that (Re-)Allocate TMSI	Secure*
Free5GC [27]	Monotonic Counter	Constant	RegistrationRequest, N1MsgNotify, CreateUEContext	N
SD-Core [26]	Go `math.rand`	Timestamp (ns)	RegistrationRequest, N1MsgNotify, CreateUEContext	N
Open5GS [38]	Static Pool (2048 TMSIs)	`/dev/urandom`	RegistrationRequest, SessionModification, Paging	Y
OAI-5G [43]	Monotonic Counter	Constant	SecurityModeComplete	N
OAI-LTE[44] w/ Magma MME[3]	libc `rand()`	Timestamp (s)	RegistrationRequest, IdentityResponse, AttachAccept	N
NextEPC [33]	Static Pool (16,384 TMSIs)	`/dev/urandom`	First NAS Message from UE	Y

* Secure in that TMSIs are unpredictable

First, the attacker begins eavesdropping on unciphered cellular communications between UEs and the base stations associated with the given AMF. The attacker can now monitor when TMSIs expire over the network since they are sent in plaintext. TMSIs mask several bits from the generated random value to ensure the resulting TMSI is well-formed. Therefore, an attacker may not easily observe them in an unbroken ordered sequence. However, Argyros and Kiayias [14] demonstrated that internal state can still be extracted from weak PRNG algorithms even when dealing with non-consecutive, truncated outputs using off-the-shelf hardware. As they have demonstrated the feasibility of such an attack, an attacker can observe TMSI values assigned by an AMF/MME over time to recover the internal state of the PRNG. Therefore, using these insecure PRNG algorithms enables an attacker to reliably predict future TMSI values just by observing present TMSIs.

Having broken the PRNG that the core uses to generate TMSIs, the attacker still must guess which is the next TMSI to be used. Therefore, the attacker associates a UE they control to obtain a new TMSI from the network. Afterward, they can iterate through the list of predicted TMSIs until they reach the value of their TMSI. The attacker now knows that the next predicted TMSI will be the next TMSI allocated when an association is observed. The attacker may now chronologically order UE associations it observes by monitoring unciphered cellular communications and extracting old TMSIs within the association messages. Each subsequent association is determined to be assigned the next predicted TMSI by the attacker, so they map old TMSI values observed to newly predicted TMSI values. A successful attacker can track the subscribers associated to *chains* of TMSI values. Such TMSI linkability fundamentally violates core architectural assumptions of the cellular network, and allows for real-time tracking of subscriber locations [37].

5.4 TLS Failures

CellCrypt additionally identified multiple failures in TLS verification. Specifically, when initiating a TLS connection between two NFs in the core, each party did not verify the identity or validity in the TLS certificate. This generally results in insecure communication among NFs in the core network.

Disabled Certificate and Hostname Verification: *free5GC, SD-Core, OAI-4G, and OAI -5G do not verify TLS certificates or hostnames.* CellCrypt finds TLS verification is not enforced in free5GC, SD-Core, and OAI. In all of these code bases, there is a configuration setting to turn on TLS or HTTP2. Therefore, even when TLS is configured, the code bases enforce it insecurely. This ensures that confidentiality and integrity between each NF in the core is entirely compromised regardless of configuration settings.

```
innerHTTP2Client = &http.Client{
        Transport: &http2.Transport{
            TLSClientConfig: &tls.Config{
                InsecureSkipVerify: true,
            ...
 func CallAPI(...) (*http.Response, ...) {
    ...
    if request.URL.Scheme == "https" {
        return innerHTTP2Client.Do(request)
    ...
```

Listing 2: The InsecureSkipVerify parameter in Go's HTTP library disables certificate validation.

```
static connection_t *connection_add(...) {
  ...
  if (client->scheme == OpenAPI_uri_scheme_https) {
    if (client->insecure_skip_verify) {
      curl_easy_setopt(conn->easy, CURLOPT_SSL_VERIFYPEER, 0);
      curl_easy_setopt(conn->easy, CURLOPT_SSL_VERIFYHOST, 0);
    }
    ...
```

Listing 3: In C++ code bases, disabling certificate and hostname validation each required an adjustment to cURL settings. Open5GS guards this setting with a configuration, other code bases do not.

We find that in both Go and C++ code bases, hostname verification, and certificate verification are disabled in all instances. In Go, the `InsecureSkipVerify` option disables both certificate and hostname verification. In C/C++ code bases, we identify two separate calls to cURL options that disable certificate and hostname verification individually. The code bases typically disable these options in one code location, which disables hostname verification and certificate validation in all HTTPS operations. In Figure 3 we show Open5GS disabling certificate validation. Notably, Open5GS is the most secure in this regard, as it contains a configuration option to adequately enforce TLS certificate and hostname verification. Other code bases have no such configuration option and do not validate TLS certificates regardless of configuration.

It is worth noting that the 3GPP standardizes TLS as optional; however, if enabled in free5GC, SD-Core, or OAI it is *inherently* insecure. A provider intending to use TLS will be misled into believing it is secure when configuring it. Additionally, various security-sensitive parameters are passed over TLS, such as cryptographic intermediaries in 5G-AKA. For example, the UDM sends the AUTN, xres*, and K_{AUSF} to the AUSF over the channel where TLS certificates are not verified. The K_{AUSF} is particularly important as session keys for UEs are derived from this key. An attacker with this key can perform session hijacking or an on-path attack.

5.5 Additional Findings

Constant Unused Signing Key: *SD-Core contains unused non-test code that is exposed via HTTP and signs an OAuth token with a constant string.* While unused, this code is exposed over an HTTP API from the NRF. SD-Core is a fork of an earlier version of free5GC. free5GC has removed this code upstream. We do not believe an attacker can exploit this functionality. However, we highlight this finding as an unexpected misuse of cryptography in the core.

CVE-2022-46397: Vector Packet Processing (VPP) is a framework for developing networks that provide features of switches and routers [25]. OAI includes multiple versions of the User Plane Function (UPF) network function. The UPF creates an IPsec tunnel from the UE to the Internet, and all normal user traffic travels through the UPF. One of these UPF implementations uses VPP as the foundation. In this UPF, they pin their dependency to a version with known CVE-2022-46397. CellCrypt also discovered this CVE during analysis. Since all Internet traffic passes through the UPF from a UE, this channel can handle highly sensitive information. Consequently, predicting IV values allows an attacker to get one step closer to reading sensitive user-plane traffic.

Deprecated Algorithms: *Many code bases use insecure hash algorithms; however, they are part of the 4G and 5G standard.* MD5 and SHA1 are algorithms in the IP Multimedia Subsystem (IMS) and Session Initiation Protocol (SIP). SIP is responsible for initiating and managing communications including VoLTE and VoNR. While these hash algorithms are approved by the SIP and 3GPP standards, they have long been known to be insecure. We understand the reason for including these algorithms, especially for backward compatibility. However, we strongly encourage the 3GPP to deprecate MD5 and SHA1 as they are insecure.

6 RELATED WORK

Cryptography Misuse: Piccolboni et al. [47] proposed CryLogger to detects cryptographic misuse in Android applications. Rahaman et al. [48] also target Java applications with their tool CryptoGuard. Brumley et al. [23] performed an empirical evaluation of misuse in Android applications with their tool CryptoLint. None of our target cores are Java-based, and therefore we could not use them for our analysis. Li et al. [39] presents one of the first analysis tools with an exhaustive ruleset for misuse in Go applications. Unfortunately, their tool was not available for use.

Ami et al. [12] evaluate existing analysis tools to see what instances of cryptographic misuse they fail to identify. Instead of an analysis tool, they create a test bed that mutates instances of misuse to see if tools properly flag them. Braga et al. [19] also perform an evaluation of existing tools and create a list of rules to check.

Cellular Security: Prior work analyzed LTE security from both attacker and defender viewpoints [17, 35, 46, 52]. Work on LTE focuses on the radio access network (RAN). LTE is primarily proprietary closed-source software running on specialized hardware. Furthermore, the RAN is only a small part of the cellular network.

5G has seen comparatively less investigation. Multiple works have studied 5G security [8–10, 50, 57]. Of particular relevance, Basin et al. [16] and Cremers et al. [21] perform a protocol analysis on the 5G AKA using Tamarin. They discover theoretical vulnerabilities relating to key derivation and storage. Prior work has analyzed security based solely on the 3GPP standards. Our work is one of the first to look at code implementations.

TMSI Attacks: Prior work has discussed TMSI attacks and vulnerabilities [13, 32, 53]. Hussain et al. [31] present a novel TMSI linkability attack called ToRPEDO. They use the linkability from ToRPEDO to enable deanonymization and IMSI brute force attacks. Rupprecht et al. [53] also propose a TMSI linkability attack. Their attack allows a passive adversary to link TMSIs to RNTIs, another identifier. Linking these two identifiers allows the mapping of a device to its radio session. Most of these works also discuss TSMI linkability attacks can enable further attacks. For example, deanonymization and IMSI hijacking can all be enabled from TMSI linkability.

7 CONCLUSION

Open-source cellular cores are being deployed in private networks as part of proprietary offerings. The significant engineering efforts leading to their maturity will undoubtedly make them a foundation for cellular ecosystems moving forward. To this end, this paper has studied the use of cryptography and secure randomness in open-source cellular cores. We designed a set of 13 CodeQL static program analysis rules and studied 7 open-source celluar cores written in both C/C++ and Go. In applying these rules to the open-source cores, we identified two significant security vulnerabilities, finding that multiple cores (1) predicably generate TMSIs and (2) improperly verify TLS certificates, even when TLS is enabled in the software configuration settings. We also identified several additional findings, including an (unused) hard-coded cryptographic key, a library dependency pinned to a version with a cryptography flaw, and wide-spread use of MD5 and SHA1 for backwards compatibility. We hope that our findings and framework will help secure deployments of these open-source cores as well as downstream proprietary projects that depend on them.

ACKNOWLEDGMENTS

This work is supported in part by NSF grants CNS-2054911, CNS-2055014, CNS-1933208, and an NSF Graduate Research Fellowship. Any findings and opinions expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] 2023. All-in-One. 5G. https://www.rakwireless.com/en-us/5g
[2] 2023. Firecell. https://firecell.io/
[3] 2023. Magma – Linux Foundation Project. https://magmacore.org/
[4] 3GPP. 2020. *3G security; Security architecture.* Technical Specification (TS) 33.102. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2262 Version 16.0.0.
[5] 3GPP. 2020. *3G security; Security architecture.* Technical Specification (TS) 33.105. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2264 Version 16.0.0.
[6] 3GPP. 2020. *Security architecture and procedures for 5G System (5GS).* Technical Specification (TS) 33.501. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3169 Version 15.4.0.
[7] 3GPP. 2023. *Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3.* Technical Standard (TS) 24.301. https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1072
[8] Ijaz Ahmad, Tanesh Kumar, Madhusanka Liyanage, Jude Okwuibe, Mika Ylianttila, and Andrei Gurtov. 2017. 5G Security: Analysis of Threats and Solutions. In *Proceedings of the IEEE Conf. on Standards for Communications and Networking.*

[9] Ijaz Ahmad, Tanesh Kumar, Madhusanka Liyanage, Jude Okwuibe, Mika Ylianttila, and Andrei Gurtov. 2018. Overview of 5G Security Challenges and Solutions. *IEEE Communications Standards Magazine* 2, 1 (2018).

[10] Ijaz Ahmad, Madhusanka Liyanage, Shahriar Shahabuddin, Mika Ylianttila, and Andrei Gurtov. 2018. Design Principles for 5G Security. *A Comprehensive Guide to 5G Security* (2018).

[11] Mujtahid Akon, Tianchang Yang, Yilu Dong, and Syed Rafiul Hussain. 2023. Formal Analysis of Access Control Mechanism of 5G Core Network. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM.

[12] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. 2013. Why Crypto-Detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[13] Myrto Arapinis, Loretta Ilaria Mancini, Eike Ritter, and Mark Ryan. 2014. Privacy through Pseudonymity in Mobile Telephony Systems. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.

[14] George Argyros and Aggelos Kiayias. 2012. PRNG: Pwning Random Number Generators. In *Proceedings of BlackHat*.

[15] Elaine Barker. 2020. *Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. Technical Report SP 800-175B Rev. 1. National Institute of Standards and Technology.

[16] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. 2018. A Formal Analysis of 5G Authentication. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[17] Anastasios N Bikos and Nicolas Sklavos. 2012. LTE/SAE Security Issues on 4G Wireless Networks. *IEEE Security & Privacy* 11, 2 (2012).

[18] Ravishankar Borgaonkar, Lucca Hirshi, Shinjo Park, Altaf Shaik, Andrew Martin, and Jean-Pierre Seifert. 2017. New Adventures in Spying 3G & 4G Users: Locate, Track, Monitor. In *Proceedings of Blackhat*.

[19] Alexandre Braga, Ricardo Dahab, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. 2017. Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*.

[20] cplusplus. 2023. rand. https://cplusplus.com/reference/cstdlib/rand/.

[21] Cas Cremers and Martin Dehnel-Wild. 2019. Component-based Formal Analysis of 5G-AKA: Channel Assumptions and Session Confusion. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.

[22] Filippo Dolente, Rosario Giuseppe Garroppo, and Michele Pagano. 2024. A Vulnerability Assessment of Open-Source Implementations of Fifth-Generation Core Network Functions. *Future Internet* 16 (2024).

[23] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An Empirical Study of Cryptographic Misuse in Android Applications. In *Proceedings of the ACM Conf. on Computer and Communications Security (CCS)*.

[24] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 2012. Why Eve and Mallory Love Android: An Analysis of Android SSL (in) Security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM.

[25] FD.io. 2023. What is the Vector Packet Processor (VPP) — The Vector Packet Processor v24.02 documentation. https://s3-docs.fd.io/vpp/24.02/

[26] Open Networking Foundation. 2023. SD-Core. https://opennetworking.org/sd-core/.

[27] free5GC Project. 2023. free5GC. https://www.free5gc.org/.

[28] GnuTLS. 2023. 5.13 Random Number Functions. https://gmplib.org/manual/Integer-Random-Numbers.

[29] Zvi Gutterman, Benny Pinkas, and Tzachy Reinman. 2006. Analysis of the Linux Random Number Generator. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[30] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin Butler. 2022. FIRMWIRE: Transparent Dynamic Analysis for Cellular Baseband Firmware. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.

[31] Syed Rafiul Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. 2019. Privacy Attacks to the 4G and 5G Cellular Paging Protocols Using Side Channel Information. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*.

[32] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 2019. 5GReasoner: A Property-Directed Security and Privacy Analysis Framework for 5G Cellular Network Protocol. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. ACM.

[33] NextEPC Inc. 2019. NextEPC. https://nextepc.com/.

[34] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. 1998. Cryptanalytic Attacks on Pseudorandom Number Generators. In *International Workshop on Fast Software Encryption*. Springer.

[35] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. 2018. Touching the Untouchables: Dynamic Security Analysis of the LTE Control Plane. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[36] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2019. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic Apis. *IEEE Transactions on Software Engineering* 47, 11 (2019).

[37] Denis Foo Kune, John Koelndorfer, Nicholas Hopper, and Yongdae Kim. 2012. Location Leaks on the GSM Air Interface. *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)* (2012).

[38] Sukchan Lee. 2023. Open5GS. https://open5gs.org/.

[39] Wenqing Li, Shijie Jia, Limin Liu, Fangyu Zheng, Yuan Ma, and Jingqiang Lin. 2022. CryptoGo: Automatic Detection of Go Cryptographic API Misuses. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[40] Junrong Liu, Yu Yu, François-Xavier Standaert, Zheng Guo, Dawu Gu, Wei Sun, Yijie Ge, and Xinjun Xie. 2015. Small Tweaks Do Not Help: Differential Power Analysis of Milenage Implementations in 3G/4G USIM Cards. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer.

[41] MITRE. 2023. CWE-295: Improper Certificate Validation. https://cwe.mitre.org/data/definitions/295.html.

[42] MITRE. 2023. CWE-297: Improper Validation of Certificate with Host Mismatch. https://cwe.mitre.org/data/definitions/297.html.

[43] OpenAirInterface.org. 2023. OpenAirInterface | 5G Software Alliance for Democratising Wireless Innovation. https://openairinterface.org/.

[44] OpenAirInterface.org. 2023. OpenAirInterface Software Alliance. https://github.com/openairinterface.

[45] OWASP. 2023. Testing for Weak SSL TLS Ciphers Insufficient Transport Layer Protection. https://owasp.org/www-project-web-security-testing-guide/stable/4-Web_Application_Security_Testing/09-Testing_for_Weak_Cryptography/01-Testing_for_Weak_Transport_Layer_Security.

[46] Yongsuk Park and Taejoon Park. 2007. A Survey of Security Threats on 4G Networks. In *IEEE Globecom workshops*.

[47] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P Carloni, and Simha Sethumadhavan. 2021. Crylogger: Detecting Crypto Misuses Dynamically. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[48] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High Precision Detection of Cryptographic Vulnerabilities in Massive-Sized Java Projects. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*.

[49] J. Rizzo and T. Duong. 2011. Browser Exploit Against SSL/TLS. https://packetstormsecurity.com/files/105499/Browser-Exploit-Against-SSL-TLS.html.

[50] David Rupprecht, Adrian Dabrowski, Thorsten Holz, Edgar Weippl, and Christina Pöpper. 2018. On Security Research Towards Future Mobile Network Generations. *IEEE Communications Surveys & Tutorials* 20 (2018).

[51] David Rupprecht, Kai Jansen, and Christina Pöpper. 2016. Putting {LTE} Security Functions to the Test: A Framework to Evaluate Implementation Correctness. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*.

[52] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. 2019. Breaking LTE on layer two. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[53] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. 2019. Breaking LTE on Layer Two. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*.

[54] Sanjeev Saharan and Jitender Kumar. 2017. Exploiting GSM Vulnerabilities: An Experimental Setup And Procedure To Map TMSI And Mobile Number. *International Journal of Advanced Research in Computer Science* 8, 5 (2017).

[55] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. 2014. Modelling Analysis and Auto-Detection of Cryptographic Misuse in Android Applications. In *Proceedings of the IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC)*.

[56] srsRAN Project. 2023. srsRAN Project - Open Source RAN. https://www.srsran.com/.

[57] Yanbin Sun, Zhihong Tian, Mohan Li, Chunsheng Zhu, and Nadra Guizani. 2020. Automated Attack and Defense Framework Toward 5G Security. *IEEE Network* 34, 5 (2020).

[58] Patrick Traynor, Patrick McDaniel, and Thomas La Porta. 2008. *Security for Telecommunications Networks*. Vol. 40. Springer Science & Business Media.

[59] VulBusters. 2023. Exploring Go's math/rand. https://medium.com/@vulbsters/exploring-gos-math-rand-b4ef0e841591

[60] David Wagner, Bruce Schneier, and John Kelsey. 1997. Cryptanalysis of the cellular message encryption algorithm. In *Proceedings of the International Cryptology Conference (CRYPTO)*. Springer.