2018 IEEE Symposium on Security and Privacy Workshops

Redesigning Secure Protocols to Compel Grammatical Compliance Checking

Keith Irwin
Department of Computer Science
Winston-Salem State University
Winston-Salem, NC 27103
Email: irwinke@wssu.edu

***Abstract*—A frequent goal of LangSec research is to demonstrate the benefits of having protocol definitions whose messages can have their grammar checked to ensure that they are well-formed prior to any other processing. It is generally taken as a given that although we can design protocols which enable this checking, we cannot compel implementers of these protocols to actually carry out these checks. In this paper we demonstrate that it is possible to modify the protocols through the use of encryption such that the implementer is essentially required to do the checking if they wish their implementation to interoperate with other implementations without errors. Crucially, this will be the case even when they are only sending and receiving well-formed messages, thus transforming the silent vulnerability of unchecked messages into an obvious error. In specific, we demonstrate how to do this for checking whether or not messages or portions thereof belong to a specified regular language and whether or not they belong to a specified context-free language.**

***Index Terms*—Network Security, Secure Protocols, Formal Languages.**

I. Introduction

A frequent aim of the LangSec community is to encourage the creation and use of network protocols and file formats for which it is possible to formally validate inputs before processing them. This is done to prevent a range of attacks which are based on sending malformed messages or files to programs which intermingle processing and parsing, resulting those programs behaving in ways which the original programmers would find both unexpected and undesirable.

Having the formats for which validation is at least possible is a clear step in the right direction. However, having that possibility does not necessarily imply that the input will actually be validated before processing. It is an unfortunate fact that in ordinary network protocols, it is easier to omit security checks than to include them. There have been a range of omitted checks which have caused difficulties from failing to properly validate signatures to overlooking bounds checks to failing to ensure that names match on delegated certificates. We have also seen, time and again, simple file formats and protocols whose messages could be formally validated before processing not be validated at all.

Properly implementing validation often requires learning how to use new tools or libraries, and it often seems easier to programmers to write their own parsing code. Thus many programs simply assume that the messages they receive are well-formed and proceed on that assumption. Although the use of tools like parser generators and regular expression libraires can often improve the program and make it simpler and more intuitive as well as providing validation, if a programmer has already written a shotgun parser, adding the validation will extra work which may be seen as unnecessary.

In practice, this sometimes means that the real-world forces which impact development such as time-pressure, budget-pressure, managing complexity, forgetfulness, and even laziness can push towards input not being validated. This is a substantial security problem because unlike other classes of bugs, security bugs are very commonly invisible to the end user. To the user encountering normal, non-attack situations, the version of the program which fails to validate its input at all looks exactly like the version which validates its input completely and correctly. This is because in normal operation when no one is attacking them, both versions are being given valid inputs. It is only once someone compromises the program that the user suddenly becomes aware of the difference.

At first glance, this would simply seem an inevitable fact of security about which we can do nothing. But with some cleverness, it turns out that we can design security protocols which address even this problem. Our previous work [1] described a method by which we can build security protocols which have the property that implementations which follow these modified protocols are simpler when they implement the security checks and more complex when they do not. However, it only describes one particular type of check and this paper introduces two more.

The basic technique which is used is to have a special function G which can be easily calculated during the process of carrying out the security check but is very complex to calculate without doing the security check. The output of the G function is then used to encrypt some portion of the message or of a following message, thus requiring that the result of G be calculated in order to read the messages. This means that any implementation of the protocol must either do the required checks and calculate the G function in the process or find some complex way to calculate the G function without doing the checks. Because checks are usually omitted due to mistakes or time concerns rather than any desire on the part of the implementer to avoid them, this should effectively compel all implementations to carry out the checks.

DOI 10.1109/SPW.2018.00048

If the checks are omitted, then the G function will not be calculated and thus we expect that the implementation which omits the checks will be broken in obvious ways as it lacks the decryption key. This will be true even when dealing only with well-formed messages which comply with the security checks. Thus the omission of security checks will not only be obvious when attacks appear, but should be immediately clear as soon as an implementation is tried against a reference implementation or other existing implementation of the protocol. This approach is not quite a complete solution to security checking because it is possible for a program to carry out a security check and then ignore the results, but it is a further step along the path.

As we urge people to move away from protocols whose messages cannot be validated, it will be for naught if the programs which implement these protocols do not validate their messages. Thus, in this paper, we introduce techniques to create protocols which implementions of the protocols are essentially compelled to validate.

Specifically, in the scheme outlined in the previous paper, there are different G functions for different classes of security checks. The previous paper describes a G function for inequality checks but due to space limitations did not include G functions for more complex checks. In this paper we present two new G function for validating that a string belongs to a particular langauge.

The main contributions of the paper are:

- A G function for regular language membership checking (section III). This function can be used in protocols to demonstrate that an implementation has checked that a string belongs to a specified regular language by ensuring that a particular finite state automaton has accepted it.
- A G function for context-free language membership (section IV). This function can be used in protocols to demonstrate that an implementation has checked that a string belongs to a specified context-free language by demonstrating that a specified grammar accepts it.

We expect that these functions can be easily included into an existing parsing process by inclusion into regular expression parsing libraries, lexer generators, and parser generators. Thus programmers will be compelled to use appropriate tools and libraries which do the formal validation as part of the parsing process.

II. G Functions

Before we explain the new G functions in detail, it seems beneficial to explain the desired properties of G functions. In understanding these properties, it is important to note that the expectation for how G functions will be used is that their results will be incorporated into a message in the protocol as an encryption key which is used to encrypt some important portion of a message.

Often messages can be divided into one portion which is checked and a second portion which will be encrypted using the value of the G function from those checks. For example, it may be that the message header has its checks carried out, but the body just contains user data. For messages which cannot be easily separated, the G functions for the checks from one message can be used as the encryption key for the next message sent. For simplicity of description, we are going to assume the first case where a key is used within the same message (although not on the same part which is checked).

This means that in normal operation the sender will create a message, do the security checks while calculating the value of G function for the message they are sending so that they will have the needed encryption key, and then use that key to encrypt a portion of the message. They do this not because they would normally need to check any security properties of the message they just generated but because the receiver will need to do those checks to ensure that the protocol is secure. When the receiver receives the message, they will carry out the required checks while also calculating the value of the G function. The result of that calculation can then be used to decrypt the encrypted portion of the message and the receiver can proceed.

Each G function is specific to some check which should be carried out on a message. There are three properties for G functions outlined in the previous paper [1]. To this list we add one additional property which was implied by the paper but not explicitly listed.

1) When the check fails, G should be undefined.
2) When the check succeeds, G should have a large range of possible values.
3) When the check succeeds, the value of G should be difficult to predict without doing the check.
4) There should not be a small mistake that the programmer could make in implementing G which would result in the correct value of G when the check succeeds and a predictable value of G when the check fails.

There are two goals behind these three properties. The first goal is that any implementation of a protocol which uses a given G function must do one of three things in order to interoperate with other implementations of the protocol: properly implement the check, brute force the key, or have complex code which computes the value of G without doing the check. The second property above exists to make the second option unattractive to implementers. Likewise, the third property above exists to make the third option unattractive.

The second goal of the three properties is to make it unlikely that an attacker could craft a message where the check fails but where the implementation fails to recognize this. The first property means that when the check fails there will be no value of G to compute. The fourth property should mean that small mistakes in implementing the check won't open up potential attacks. An example of a situation where the fourth property might not be present would be if a G function included the index where a particular value occurred on a list. Even though that quantity is undefined when the value fails to appear in the list, many implementations return default or failure values like -1. Attackers would be able to recognize this and adjust their encryption keys to match.

An easy way to construct a G function with these properties is to use the Z function. This is originally defined in [1], however there was an error in that definition. The corrected version of the Z function is included here.

The Z function is a function which is undefined for the value of 0, has a range of values otherwise, and which is designed such that we can be fairly certain that there is no simple implementation error which will produce correct values for valid inputs and a value for an invalid input (in this case, only 0).

The Z function assumes that the protocol specification chooses two positive constants b and z_b and a hash function H. The Z function is designed so that it has to iterate through one or two different cases to get from the input to the number b generating a randomized hash along the way. Each of these cases will approach their target rapidly ($O(log(b))$) but with a number of repetitions which is hard to predict exactly.

The Z function is defined as follows:

$$Z(i) = \begin{cases} z_b & \text{if } i = b \\ H(Z(2 * i)) & \text{if } 0 < i < b \\ H(Z(\left\lfloor \sqrt{ib} \right\rfloor)) & \text{if } i > b \end{cases}$$

A proper implementation of a Z function will check if it has been passed a zero and return this as an error as Z is undefined for 0. However, an implementation which omits this check might pass that 0 into one of the two recursive cases (due to falling through to an else or not checking both parts of the condition for the first case) However, in both cases if given a 0 as input, they will be infinite loops thus ensuring that no value will be returned in the event that an implementation fails to check .

This does have the potential to turn a silent error caused by a failure to check for malformed messages into a denial-of-service attack. There may be some cases where this could be worse such as systems where reliability problems could prove fatal. However, in most systems, we argue that an obvious shutdown of a system is preferable to a silent exploitation of that system.

To use the Z function to create a G function, you create an expression which evaluates to zero when the check fails and non-zero when the check succeeds. Because the Z function is undefined for $Z(0)$ and would loop infinitely if implemented without an explicit check for 0, G functions defined using Z generally have the three properties so long as the expression inside the Z function produces a wide range of non-zero values for different messages.

When a protocol specifies that the messages it uses should be members of a particular language, there are still a number of things which might go incorrectly which would allow implementations to accept messages which do not conform to the specified language. The simplest is that the language may not check the validity of the message at all. Another possibility is that they might do informal validation. This could take the form of writing custom code which tries to screen out abnormal messages or recognize normal ones without formal checking. Or it could also take the form of a "shotgun parser" which parses as it processes the message and, as a result of the complexity of logic, misses some cases.

Another possibility is that they could use code which they believe will do formal validation, but which is insufficient or incorrect. In general, when using a tool which is less powerful than what is required to match the given language the way to successfully accept messages which are known to be valid is to be overly permissive in what you accept. For example, as is well known, it is impossible to write a regular expression which accepts exactly the language of all strings of the form $a^n b^n$. But it is trivially easy to build a regular expression which accepts every string from that language as valid. The regular expression $a^* b^*$ is one simple solution. In that example, a programmer might feel that the implementation is correct. After all, every valid message is accepted. Their testing may include no invalid messages or only limited invalid messages.

For example, there are sometimes pattern-matching libraries (a good example of this is the built-in pattern library in Lua) which are not as expressive as regular expressions. A programmer might use one of these to attempt to match a regular language and accidentally create code which also accepts some messages which are outside the intended language. Or a programmer could use a regular expression or a small group of regular expressions to validate messages which should be checked against a context-free language.

They also might attempt to implement their own regular expression validator or their own parser and make mistakes in the process of doing so. For example, their code might always accept and that would not be noticed when only testing with well-formed expressions.

We would like our system to expose these mistakes. Specifically, if checks have not been implemented, this should be obvious to the user because the program will lack the knowledge of the G function value used to decrypt parts of the message (or possibly the next message). Or, if the checks are performed incorrectly, the wrong key should be derived some of the time (even when the message is actually valid). When the key is missing or the wrong key is derived, this means that the encrypted data cannot be decrypted. It is expected that this difficulty will be obvious to the user either in the form of scrambled data or other errors resulting from messages or portions of messages not being decrypted.

III. Regular Language Checking

In this section we describe the construction of a G function to ensure that an implementation has checked regular language membership. There are several constants which will be required in this G function which we assume will be chosen by the protocol designer and then fixed in the protocol specification.

The protocol should also specify a particular finite-state automaton to carry out the recognition of the messages. We would like the results of our G function to validate several properties:

- That the string being recognized is the one from the message.
- That the final state is an accept state.
- That the transitions that the implementation takes are valid.
- That the correct FSA is in use

The first we will establish by ensuring that the data is mixed into the G function by using information about the particular transactions taken in our G function. The second and third we will establish by ensuring that the G function is undefined in the event that either any invalid transition is taken or if the final state is not an accept state. The fourth we will tackle through our labeling procedure which we will describe after we define G since its function is somewhat orthogonal to the other parts of the process. We must ensure these properties while also ensuring that there are a range of different possible values when the finite-state automaton accepts the string.

In defining G we assumed that we have a finite state automaton (FSA). In general, we define a finite-state automaton in the standard way as having an input alphabet of Σ, a set of states S, initial state $s_0 \in S$, a state transition partial function $\delta : S \times \Sigma \to \mathcal{P}(S)$, and a set $F \subset S$ of accept states.

Now we are going to tackle the second property above by defining a function A which takes in a deterministic finite state automata, $\langle \Sigma, S, s_0, \delta, F \rangle$, and a string, m, and returns an integer value for use in defining G. An important property of A is that A will be zero if the final state is not an accept state and will otherwise be a non-zero number which reflects the transitions which have been taken up until the final accept state.

In order to do this, we first label every state with a unique positive integer (in a way specified in the protocol) with the property that accept states are labeled with even numbers and non-accept states are labeled with odd numbers. So we define a function b which represents this label such that $b(s)$ is the label on state s.

When we use our finite state automaton to recognize our message m, this results in it transitioning through a series of states, $s_0, s_1, \ldots, s_n$.

We then define a function $a(i)$. This function will fold in the value of the labels of the states thus connecting the input data to the value, but it will also have one bit of information about the last previous state to allow us to know if the sequence ends in an accept state.

$$a(i) = \begin{cases} b(s_0) & i = 0 \\ 2a(i-1) + b(s_i) & i > 0 \end{cases}$$

Then we let $A(\Sigma, S, s_0, \delta, F, m) = a(n) \cdot j$ where j is the largest integer such that 2^j divides $a(n)$. If s_n is not an accept state then $a(n)$ will be odd and thus j will be 0. If s_n is an accept state than $j \geq 1$ and as $b(s) > 0 \forall s \in S$, it follows that $a(n) > 0$. We will use this part to achieve the property that only sequences of states which end in an accept state will be valid.

Note that the size of a will be $2n + lg(b(s_0))$ in the worst case. This is somewhat inconvenient, and we suspect that the inputs could be processed differently to achieve the same effective results, but for the moment, no such alternate approach has occured to us. This refinement will be reserved for future work.

Next we address the third property, that only valid transitions are accepted. Specifically, we are concerned that transitions only occur in situations where δ is defined. If δ is undefined for a certain combination of symbol and state, it is possible that an implementation could accidentally transition to a default state of some sort or accidentally follow the first or last transition on its list.

For this property, we define a function V which takes in a deterministic finite state automaton, $\langle \Sigma, S, s_0, \delta, F \rangle$, and a string, m, and returns an integer value for use in defining G. The property of V is that it will be 0 if any invalid transitions are taken.

To cause this to occur, we label each state and each symbol in the alphabet with a unique prime number. We represent this as a function $p(x)$ where $x \in S \bigcup \Sigma$.

Next, for each state, we label it with a new number we calculate which we represent with the function $v(s)$. We calculate v by multiplying $p(s)$ by the $p(\sigma)$ of each $\sigma \in \Sigma$ which is not a valid outgoing transition from that state. Formally, we define a helper function $h(s, \sigma)$ as so:

$$h(s, \sigma) = \begin{cases} 1 & \delta(s, \sigma) \in S \\ p(\sigma) & \delta(s, \sigma) \text{ is undefined} \end{cases}$$

Then we use that to define $v(s)$ as follows:

$$v(s) = \prod_{\sigma \in \Sigma} h(s, \sigma)$$

Then when our finite-state automaton processes m, m corresponds to a set of symbols $\sigma_1, \sigma 2, \ldots, \sigma_n$. This causes transitions through a set of states $s_0, s_1, \ldots, s_n$. We thus define V as follows:

$$V(\Sigma, S, s_0, \delta, F, m) = \prod_{i=0}^{n-1} v(s_i) \bmod p(\sigma_{i+1})$$

What this does, in short, is it labels each state with the product of its own label and all the labels of each symbol which does not transition out of it. Then we take this product and take it modulo the chosen symbol. Because these numbers are all prime, that will be zero when and only when the symbol whose transition was taken is not a part of a defined outgoing transition.

Note that $v(s_i) \bmod p(\sigma_{i+1})$ can be calculated by calculating

$$v(s_i) \bmod p(\sigma_{i+1}) = \\ (\prod_{\sigma \in \Sigma} h(s, \sigma) \bmod p(\sigma_{i+1})) \bmod p(\sigma_{i+1})$$

And thus this can potentially be computed without needing arbitrary precision integer arithmetic. Instead, we can simply maintain a bit vector for each state and a listing of which primes have been chosen. Then we can calculate $v(s_i) \bmod p(\sigma_{i+1})$ by doing repeated modulo multiplication.

To get the V function for the whole path, we then take the v function for each state and multiply them all together. Thus if any one was an invalid transaction, V will yield zero.

Both parts finished, we can now define

$$G_{FSA}(\Sigma, S, s_0, \delta, F, m) = Z(A(\Sigma, S, s_0, \delta, F, m) \cdot V(\Sigma, S, s_0, \delta, F, m))$$

where A and V are defined as above and Z is the function from the previous work which produces an undefined result on 0 and a highly differing result for all other inputs.

A. Whole FSA Dependence

Our fourth property we wished to have above is that we wanted to make sure that we were using the correct FSA. Our worry is that rather than implementing the FSA specified in the protocol, an implementor would accidentally leave out some states or transitions or include spurious states or transitions, but do so in a manner that common messages would still be correctly recognized because the portion of the FSA exercised by common messages was correct.

In order to do this, we are going to introduce a procedure for giving the nodes different labels for each message. This is somewhat expensive, and, as such, could be omitted in the event that the protocol designer thinks it unlikely that implementers might use the wrong finite state automaton.

If this procedure is in use, before we parse the message, we first generate the labels represented by the function b that we used when calculating the function A above. Specifically, we want to fold data from the whole FSA into the labeling process so that if there is a variance in the FSA this would be reflected as mistakes in the b labels which would result in an incorrect A value a reasonable percentage of the time, even for well-formed messages, thus causing the value of G to not match and the implementation thus to appear to be obviously broken.

To begin the process when preparing a message to be sent, we generate a random or psuedo-random seed, r. This seed must be included in the message itself so that an identical procedure can be carried out by the recipient.

We let D_δ be the domain of δ, that is the set of input values for which δ is defined. We consider the set of transitions to be the set $T = \{(s, \sigma, \delta(s, \sigma)) \forall s, \sigma \in D_\delta\}$ We then assume that we have some consistent deterministic ordering of T which we represent by the function $t : [0..|T| - 1] \to T$.

We assume a set of variables $b_s \forall s \in S$ which will be used to define $b(s) = b_s$. The initial value for all such variables will be undefined. We wish to label all of the node, thus we will run until all $b(s)$ is defined for all $s \in S$. Then we follow the following procedure which utilizes some hash function H which we assume to be specified in the protocol::

$h \leftarrow r$
while any b_s is undefined **do**
  $i \leftarrow i + 1$
  $(s_0, \sigma, s_1) \leftarrow t(h \bmod |T|)$
  $j \leftarrow \lfloor v/|T| \rfloor \mod 2$
  if $s_j \in F$ **then**
    $f \leftarrow 0$
  else
    $f \leftarrow 1$
  end if
  if b_{s_j} is undefined **then**
    $b_{s_j} \leftarrow 2v + f$
  else
    $b_{s_j} \leftarrow 2H(b_{s_j} + v) + f$
  end if
  $h \leftarrow H(b_{s_j})$
end while

When this procedure is used, we will select a series of transitions and for each we will effectively ensure that either its origin or destination is correct by changing the label for it using a series of hashes. Because each hash is based on the previous one, our whole path is dependent on each part being correct. Thus, with each message, we will validate $|S|$ different transitions pseudo-randomly selected. If there are mistakes in the FSA of either the sender or receiver of a message, this will result in the wrong labels some of the time which will in turn result in the wrong output of the G function. This should then lead to obviously broken behavior for well-formed messages.

IV. Context-Free Language Checking

In designing a G fuction for recognizing a message or message part which belongs to a particular context-free language, we have to narrow things down some. Mismatching G function calculations between the message sender and the message receiver will result in a communications breakdown. Obviously, this is what we desire if the checks are not being carried out or if they are being carried out incorrectly. But if there is more than one correct way to carry out a check, this is then a problem.

In context-free languages in particular, we potentially face such a problem. Even if we simply chose a uniform language, there can be more than one grammar which parses the same language. It's not uncommon to take a published grammar specification and make small tweaks which change the generated parse trees but not the accepted language in order to make accommodations to the abilities of the parser generator in use. The specification might be written with an LR(1) parser in mind but the implementor might be using an LALR(1) parser generator instead and it may well be the case that for the particular language, either can parse it.

And even with the exact same set of production rules, different parsers could generate different (but equivalent) parse trees for the same data. There may exist a complex approach to tolerating such vagueness and ensuring that the same G function is calculated in all cases, but this paper is going to take the easy way out. Instead, we require that the protocol specify not just the language, but also the grammar and specific details about the parsing algorithms which should be used to a level of detail sufficient to ensure that when parsing the language in question, the same parse tree will be generated in all cases by both the sender and receiver. We assume that there

exists some parser P such that for a message m yields a parse tree with root $P(m)$ when m is in the specified language.

In order to create the G function for recognizing context-free-languages we are actually going to leverage our work from the previous section on creating G functions for finite state automata. Although it is obviously the case that we cannot use a single finite state automaton to recognize a context-free language (unless it is also regular), we observe that we do use finite-state machines in the normal process of recognition. In practice, we use FSAs in the lexical analysis portion to generate the lexemes which are then parsed by the context-free language parser. But beyond that, each individual production rule can actually be expressed as an FSA with lexemes and production symbols as its alphabet.

Thus when we wish to validate that the parse tree is correct relative to some grammar, we can treat it as a tree of different FSAs and validate them the same way we would FSAs.

We could actually calculate a G function for each FSA separately and then combine them using XOR in order to find the key. But a more efficient approach would be to multiply the results of the helper functions at each step.

In the previous section, we defined $G_{FSA}(\Sigma, S, s_0, \delta, F, m) = Z(A(\Sigma, S, s_0, \delta, F, m) \cdot V(\Sigma, S, s_0, \delta, F, m))$ and along with that defined the functions A and V. A is zero only when the FSA ended not on an accept state and V is zero only when the FSA had invalid transitions.

Each node in the parse tree is either a leaf node which represents a lexeme or is a non-leaf node derived from some production rule. In either case, there is an associated finite state automaton. We can also consider there to be a list of child nodes for each node. For leaf nodes the list of child nodes will be an empty list. For each node in the parse tree, n, which has a sequence of children $c = \{c_1, c_2, \ldots, c_{|c|}\}$ and a finite state automaton, $\langle \Sigma, S, s_0, \delta, F \rangle$, we define

$$AV(n) = A(\Sigma, S, s_0, \delta, F, c) \cdot V(\Sigma, S, s_0, \delta, F, c) \cdot \prod_{i=1}^{|c|} AV(c_i)^{i+1}$$

Note that although there is no explicit base case for this recursive definition, leaf nodes have no children and, as such, $|c|$ is 0 and thus their definition does not recurse further.

We raise the $AV(c_i)$ to a power so that we ensure that both the correct children are present in the parse tree and also that they are present in the correct order. With this in place, we can define $G_{CFL}(P, m) = Z(AV(P(m))$.

This function will be undefined in the event that the parser P fails to accept m. This will be true either if the parser does not return a root node, as it should not, or if it returns something such as an incomplete or malformed parse tree. The individual finite state automata which correspond to the production rules thus serve as a check of parser correctness.

It should be noted, though, that unlike the G function for finite state automata, this function fails to cause errors when a parser is using a grammar which has differences from the correct grammar but where those differences are not exposed in the individual message being parsed. In the long run, we would like an improved G function which does have this property.

V. Related Work

In general, this is paper expands a fairly new idea, and as such, there is limited related work. There has been a fair amount of research which shares our same goals of ensuring the implementations are correct, but the others employ very different means. We present a sampling here.

There are several papers focussed on applying formal methods to verifying the correctness of implementations including carrying out needed checks. Some of them focus on model checking such as [2]. Others are more focussed on conformance testing such as [3], [4]. These are both good approaches, when they are used, but our approach has the advantage that unlike tool use, protocol compliance is generally not voluntary.

There have also been a number of papers which have focused on verifying the correctness of security protocols using model checking or theorem proving such as [5], [6], [7]. This research is orthogonal to ours as we do not test the correctness of the protocols at all, but rather just try to ensure correct implementation. However, some of these have gone beyond this into taking these formal descriptions of security protocols and automatically generating code for them [8], [9] which would also necessarily include needed checking. This is a very reasonable approach, but again, has not been widely deployed in the real world. Also, all of the existing automated code generation tools for secure protocols are restricted to a single language for output, which limits their impact. Changes to the protocol itself will cause implementation changes across all platforms.

VI. Conclusion and Future Work

In this paper we have shown how you can design protocols which should have the property that it is easier to implement a program which does the grammatical checking than one which skips it. We describe versions of the G function which can serve this role for both regular languages and context-free languages which thus covers most grammars whose use in protocols is sensible.

There is some room for improvement. As we noted in the section for the G function for context-free languages, our function does not catch the case where a program implements a variant of the correct language which overlaps for all common messages but differs for unusual ones. Our future plans include attempting to redesign it to provide this assurance.

Also, although there is a compelling case why this approach should result in protocols doing the required checks we have not validated these results experimentally. To do so, we would need to build two versions of a given protocol one with our technique and one without and then have two randomized groups of programmers implement them. Then we could

compare how frequently programmers missed checks in both cases.

We are also planning to describe formally and implement in code processes for transforming a description of a normal secure protocol and its required checks into a protocol modified to add the required G functions to assure that the checks have been completed. This modified protocol would include all required labels and other constants.

Thus, we could also then automatically generate implementations utilizing such a specification or have a standard library which can handle the validation stages for any protocol so described before handing the result over to an implementation of the unmodified protocol.

In summary, this approach has a lot of promise, but there remains a lot of work which can be done to move it towards practicality.

REFERENCES

[1] K. Irwin, "Redesigning secure protocols to compel security checks," in *Security Protocols XXIII*, B. Christianson, P. Švenda, V. Matyáš, J. Malcolm, F. Stajano, and J. Anderson, Eds. Cham, Switzerland: Springer International Publishing, 2015, pp. 22–29.

[2] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, ser. NSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251175.1251187

[3] B. S. Bosik and M. mit Uyar, "Finite state machine based formal methods in protocol conformance testing: from theory to implementation," *Computer Networks and ISDN Systems*, vol. 22, no. 1, pp. 7 – 33, 1991, 9th IFIP TC-6 International Symposium on Protocol Specification, Testing and Verification. [Online]. Available: http://www.sciencedirect.com/science/article/pii/016975529190079R

[4] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar, "Conformance testing methodologies and architectures for osi protocols," R. J. Linn and M. U. Uyar, Eds. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995, ch. An optimization technique for protocol conformance test generation based on UIO sequences and rural Chinese postman tours, pp. 427–438. [Online]. Available: http://dl.acm.org/citation.cfm?id=202035.202073

[5] D. X. Song, "Athena: a new efficient automatic checker for security protocol analysis," in *Proceedings of the 12th IEEE workshop on Computer Security Foundations*, ser. CSFW '99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 192–. [Online]. Available: http://dl.acm.org/citation.cfm?id=794199.795118

[6] D. Basin, S. Mdersheim, and L. Vigan, "An on-the-fly model-checker for security protocol analysis," in *Computer Security ESORICS 2003*, ser. Lecture Notes in Computer Science, E. Snekkenes and D. Gollmann, Eds. Springer Berlin / Heidelberg, 2003, vol. 2808, pp. 253–270.

[7] G. Lowe, "Breaking and fixing the needham-schroeder public-key protocol using fdr," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Springer Berlin / Heidelberg, 1996, vol. 1055, pp. 147–166.

[8] D. X. Song, A. Perrig, and D. Phan, "Agvi - automatic generation, verification, and implementation of security protocols," in *Proceedings of the 13th International Conference on Computer Aided Verification*, ser. CAV '01. London, UK, UK: Springer-Verlag, 2001, pp. 241–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=647770.734267

[9] D. Pozza, R. Sisto, and L. Durante, "Spi2java: Automatic cryptographic protocol java code generation from spi calculus," in *Proceedings of the 18th International Conference on Advanced Information Networking and Applications - Volume 2*, ser. AINA '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 400–. [Online]. Available: http://dl.acm.org/citation.cfm?id=977394.977464