Abstract—Software security refers to the security properties of a software system’s implementation. Various programming techniques as well as process practices can help with building a secure implementation. In this paper we explore the contribution of language engineering to solving this challenge. Language engineering refers to building, extending and composing languages using a principled approach, based on language workbenches. Specifically, in this paper we illustrate how modular extension of the C programming language can help with technical and process-related aspects of software security. To this end we rely on mbeddr, an extensible version of C developed with the JetBrains MPS language workbench. We close the paper with a discussion of the potential drawbacks of the approach and how these can be solved in the future.

I. INTRODUCTION

A secure system is one whose intended functionality cannot be disrupted by malicious attacks. Security involves many aspects including network security (e.g., not exposing open ports to a network), data security (e.g., making sure sensitive data is not exposed) and software security. Software security [1] refers to the properties of the software system itself to not be susceptible to attacks; common software security failures are exploitable buffer overruns, cross-site scripting attacks or algorithmic problems in the software that can be maliciously exploited.

Many of the software security challenges originate from careless or wrong use of programming languages [2]. C is widely used in embedded software, cyber-physical systems and the Internet of Things as well as network infrastructure. The software in these domains is also often critical in the sense that safety or security flaws can cost a lot of money, expose networks, damage physical systems or endanger lives (examples of aircraft hacking are found in [3]). Hence, addressing potential security problems in C-based software is of paramount importance for these systems.

Contribution This paper demonstrates preliminary results on how language engineering can help to achieve more secure software, and points out directions for future research. It focuses on embedded, C-based systems and relies on the JetBrains Meta Programming System (MPS) language workbench and the mbeddr embedded software development stack.

Structure In Section II we introduce language engineering, MPS and mbeddr. Section III shows examples of how language engineering can be useful for software security; it covers techniques (Section III-A) and process-related aspects (Section III-B). We discuss the approach, including its challenges and potential drawbacks in Section IV and conclude the paper in Section V.

II. LANGUAGE ENGINEERING, MPS ANDMBEDDR

Language Engineering and MPS Language engineering refers to building, extending and composing languages. The field encompasses general-purpose programming languages, but focuses mostly on domain-specific languages (DSLs) [4]. Language workbenches [5], [6] are tools for efficiently designing and implementing languages. MPS [7] is an open-source language workbench that provides comprehensive support for many aspects of language definition, including structure, syntax, type systems, transformation and generation, debugging and integrated development environment (IDE) support. MPS relies on a projectional editor which avoids parsing the abstract syntax tree (AST); instead, editing gestures directly change the AST, and the concrete syntax is rendered (“projected”) from the changing AST. This means that, in addition to text, languages can also use notations that are not parsable such as mathematical symbols, tables and diagrams [8]. Since a projectional editor never encounters grammar ambiguities, they can support a wide range of language composition techniques [9]. Traditionally, projectional editors were hard to use and were not adopted much in practice. MPS, in contrast, makes editing in a projectional editor as close to “normal text editing” as possible and also supports diff/merge on the level of the projected concrete syntax; a study we performed shows that users are mostly agreeable with the editor after a short while of getting used to it [10]. The paper also contains the techniques used by MPS to achieve this level of usability.

Embedded Software, C and mbeddr The benefits of projectional editors relative to notational flexibility and language composition have been explored in the context of embedded software engineering in the mbeddr project [11]. It provides a user-extensible version of C and ships with a set of predefined extension such as physical units, interfaces and components, state machines and
unit testing. The benefits of these extensions in terms of developer productivity, maintainability and robustness are discussed in [12]. mbeddr also supports product line variability, requirements traces and documentation. Finally, mbeddr explores the synergies between language engineering and formal verification by providing domain-specific verifications [13]. mbeddr is an open-source project licensed under the Eclipse Public License. It is currently being used in several commercial development projects and forms the basis for future controls engineering product by Siemens PL.

**Modular Language Extension** The extensions provided by mbeddr are modular, in the sense that the base language (C in our case) is extended with additional language concepts without invasively changing the base language. We call such extensions modular language extensions (MLEs); they include concrete syntax, type system, execution semantics as well as IDE support. They can also be seen as little embedded DSLs. We rely on MLEs in this paper to add security-relevant language abstractions to mbeddr. [4], [14] and [15] provide details on building MLEs in MPS.

### III. LANGUAGE ENGINEERING AND SECURITY

There are two major ingredients to software security: technique and process. Technique refers to the languages, architectures and tools used to implement a software system. Different choices may make it more or less easy to build secure systems. C is not an ideal implementation language for secure systems because programs written in C are prone to low-level mistakes that can be exploited maliciously. Its low abstraction level also makes it hard to analyze. Process refers to the practices employed to build the system [16]: reviews, education of the developers and a strong test and verification culture are ingredients of a process that can lead to more secure software. In this paper we explore the potential benefits of language engineering, MPS and mbeddr for both aspects: we discuss technique in Section III-A and process in Section III-B.

#### A. Techniques

**Code Markup and Checking** Code markup refers to annotations that are added to the code to express additional semantics. Checks associated with these annotations verify the semantics. One example is the support for physical units in mbeddr. Types and literals can be annotated with units, and the type system then checks for the correct use of units in expressions and assignments; the following code snippet shows an example.

```c
int16_t m_alt = cur->alt - prev->alt;
int8_t s = dTime - startTime - prev->time;
```

mbeddr’s units do not directly focus on security, instead they address correctness and robustness (the Mars Climate Orbiter crashed in 1999 due to a unit mismatch [17]). However, a similar approach can be used for security. Consider a system that deals with sensitive data. The data can exist in encrypted or unencrypted forms ($d_e$ and $d_u$). The software system is correspondingly structured into a non-secure and a secure part ($P_u$ and $P_e$). For security, it is crucial that no unencrypted data is in the non-secure part ($d_u \notin P_u$) and that data is encrypted as it moves from $P_u$ to $P_e$. A set of annotations on types, variables and modules can be used as the basis for data flow checks that verify these properties.

**Straightforward Language Extension** MLEs extend existing languages with additional, first-class language constructs in a modular way; they include syntax, type system, semantics and IDE support. As an example for a security-relevant language extension consider the following `trysequentially` statement (the implementation is part of the mbeddr tutorial). It can be used to address the `goto fail` bug found in Apple SSL implementation in 2014 ([https://gotofail.com/](https://gotofail.com/)).

```c
trysequentially {
    validateStep1(data, ...);
    validateStep2(data, ...);
    validateStep3(data, ...);
} on fail (errorcode) {
    handleFailedValidation(data, errorcode, ...);
}
```

`trysequentially` invokes a sequence of functions, where each of them returns an error code. If a function returns a non-zero value (i.e., reports an error), the `trysequentially` branches to the error handler. This is a higher-level version of the following C-level idiom:

```c
if (validateStep1(data, ...) != 0) goto fail;
if (validateStep2(data, ...) != 0) goto fail;
if (validateStep3(data, ...) != 0) goto fail;
fail: handleFailedValidation(data, errorcode, ...);
```

Apple’s `goto fail` had a superfluous, unconditional `goto` statement; this prevented the correct validation of SSL certificates and was a serious security issue. Since the idiomatic C code is automatically generated from the more intentional `trysequentially`, fewer coding mistakes can happen, thereby improving security. All of mbeddr’s existing MLEs (for unit tests, physical units, interfaces and components and state machines) represent direct language support for lower-level C idioms, thereby improving robustness and security by reducing the risk of mistakes in the lower level details.

**Adapting Semantics** This refers to changing the semantics of existing language constructs to make them more suitable for secure software. Consider a system that works with secret keys. Likely, the key is itself encrypted (as $k_{\text{enc}}$), but to work with the key, it has to be available in the clear as $k_{\text{clr}}$. For the software to be secure, it is important that $k_{\text{clr}}$ is kept in memory only when absolutely necessary. Consider the following code:

```c
char* encryptData(char* k_enc, char* data) {
    char k_clr[256];
    decryptKey(k_enc, k_clr);
    char* encryptedData = // encrypt with k_clr
    return encryptedData;
}
```

At the end of this function, the `k_clr` local variable becomes invalid by moving the stack pointer, but the memory allocated to the stack still contains the actual data and can potentially be exploited. To avoid this,
the memory area should actually be cleared (“zeroed”).
So, the semantics of C should be changed in the fol-
lowing way: the memory used by local variables that
leave their block should automatically be zeroed. This
can be achieved easily by a transformation that inserts
dereferencing code for each local variable at the end of a block.
 Optionally, these semantic changes can be combined with
code markup (discussed above). For example, instead
of performing the zeroing globally, it can be limited
to functions that are annotated as “secure” or to local
variables that are marked as “secure”. In embedded
systems, this may be important for performance reasons.

Exploiting the Generation Step  Most language work-
benches are generative. For example, in mbeddr/MPS the
AST of a program is translated to C text for compilation.
MLEs are transformed to C in one or several steps.
Beyond adapting language semantics, code generation
can also be used for other security-related purposes.
Below we discuss two of them.

A common attack vector are side-channel attacks [18]
which exploit non-functional properties of a system. In
a timing side-channel attack, the timing properties of a
system are exploited to reverse-engineer details about
the program’s implementation or about key material. To
prevent this, the timing behavior of a system must not
deterministically relate to the operation of the system.
To make this dependency harder to observe, random
instructions (essentially noise that does not affect the
program’s behavior) can be scattered throughout the
code. Because the final to-be-compiled source code is
generated, it is easy to automatically inject the noise with
one cross-cutting transformation, without mixing the
side-channel attack prevention concern with the business
logic of the software. Code markup can be used to select
critical areas where noise should be added.

Another example of exploiting the generation step
is the introduction of runtime checks where C does
not perform them by default. For example, a language
extension can be defined that provides length-aware
arrays or strings, and generates length/buffer checking
code. Similarly, NULL-checks can be inserted before each
pointer access to avoid segmentation faults.

Additional Constraints  An MLE can also contain
constraints that prevent the use of insecure language
constructs or library functions. For example, pointer
arithmetics can be prevented or limited, and the use of
insecure functions (such as strcpy) can be flagged as an
error. Alternatively, the constraints can report as an error
all uses of functions that are not explicitly marked as a
secure application programming interface (API).

Verification and MLE  Software verification refers to
proving specific properties of a program. In contrast to
testing, the program is not executed; instead, a verifier
analyzes the program, often performing the semantic
equivalent of an exhaustive search of possible execution
paths. Verifying security-relevant low-level C details
(such as division by zero, pointer or array access
safety) is supported directly by tools such as CBMC [19].
However, verifying application-level properties is much
harder. From a user’s perspective, the challenge is the
specification of the expected properties (often done
through code annotations or label reachability checks),
configuration of the verifier (when configured wrongly
it may not find existing property violations) and the
interpretation of the results (which are often too low-
level and detailed). In [13] we introduce an approach
called domain-specific C verification that addresses these
usability challenges. It relies on the following steps: (1)
define MLEs that imply or explicitly specify application-
level semantics (2) generate the corresponding low-level
C code including verification-specific code annotations or
labels (3) automatically invoke the verifier (4) lift the low-
level results back to the application level.
We have used this approach to verify component contracts in mbeddr,
and in [13] we describe the verification of a pacemaker.
An example that is directly relevant to software security
is given next, based on the Heartbleed bug recently
discovered in OpenSSL (http://heartbleed.com/).
In essence, the Heartbleed bug is a problem with parsing a
heartbeat packet in OpenSSL TLS. OpenSSL wrongly
assumes the well-formedness of packets received from
the network. The code below defines a simplified heartbeat
message data type. Upon receiving such a message, with
payload of a given length, OpenSSL has to send back the
payload, completing the heartbeat protocol.

```c
struct {
  uint16 payload_length;
  unsigned char *payload[payload_length];
} HeartbeatMessage;
```

Unluckily, the code above is not legal in C: it is not
possible to specify the array length using a value from
a member of the same struct. Instead, the struct must
use a pointer to the actual data. OpenSSL’s mistakes
were trusting the `payload_length` value, reading beyond
the end of the buffer referred to by the pointer, leading
to an buffer overrun, which represents a serious secur-
ity vulnerability. This problem can be detected using
verification. To enable it, we create a message with a
nondeterministically assigned data buffer:

```c
HeartbeatMessage prepareUntrustedMessage() {
  HeartbeatMessage msg;
  assign nondet msg;
  return msg;
}
```

Fig. 1 shows the mbeddr user interface after running
a CBMC-based robustness analysis. The top-right table
shows 2 of the 40+ checked properties, one of which
failed. The dereference failure happens in the selected
line containing a `memcpy` call. The bottom-right part
shows a trace that leads to the error. The nondetermi-

![Fig. 1. Running the verifier to find the Heartbleed problem.](image)
in 1 byte allocated in the payload, and the length set to 25.

An MLE could also be used to avoid such problems in the first place: a native message type could be defined that enforces consistency between a declared size and the buffer. Associated serialization and deserialization functions can be generated and can enforce this consistency.

B. Process

Better Abstraction, Simplified Review  The right choice of abstractions in the code can simplify the code review process. For a review to be productive, it is important that the code can be explained and understood easily. The more directly the code represents relevant domain abstractions, the more productive the review process becomes. For example, reviewing the `trysequentially` extension can be more effective than the review on the level the corresponding C code\(^1\).

Better Notation, Simplified Review  Beyond just suitable abstractions, suitable notations are also essential because they can more directly resemble established notations in the domain, or because a particular notation reveal certain problems in the code. Consider mbeddr’s state machines. While the abstraction “state machine” is already a significant improvement over its encoding as switch statements, the textual notation can still be improved to make review even easier. Fig. 2 shows a state machine represented as text and as a table; we are also working on a graphical notation. Another example for an easily-readable notation is given in Fig. 3.

\(^1\)This assumes that all involved parties know the semantics of the `trysequentially` extension. However, this is a reasonable assumption in a team that develops software together.

Tracing  Code reviews are done to ensure the correctness of the code (verification), but also to establish the code’s correspondence to the original requirements (validation). For this to be effective, the relationship between a piece of code and its associated requirements must be clear. Requirement tracing [20] addresses this problem by establishing explicit links between (parts of) implementation artifacts and particular requirements. In mbeddr, a requirements trace can be attached to any program node [21] (supporting tracing on any level of granularity) expressed in any language. Fig. 4 shows an example. If the code review should be driven by the requirements, navigation from a requirement to the traced program nodes is possible via MPS’ Find Usages support as well as dedicated trace reports (see below).

Expressing Security Requirements  mbeddr ships with a requirements language [21]. Each requirement is specified with an ID, a short summary, tags, and a prose description. However, just like mbeddr C, the requirements language is extensible. For example, a classification scheme can be added that classifies requirements according to their security impact. Alternatively, requirements themselves can be traced to a set of overall security guidelines. Assessments in mbeddr are customizable reports over a model. They can be used to verify that every section of code is traced to a requirement (code for which there is no requirement is a potential attack vector), or that every security requirement has at least one trace. An example of an mbeddr assessment is shown in Fig. 7.

Direct Code Review Support  mbeddr supports track-
relevant to this paper (the other dimensions are largely unaffected by the MLEs). Incrementally adding MLEs to C is a direct implementation of the Abstraction Gradient: the abstraction level can be increased incrementally if and when it makes sense. The user is not forced to encode everything in either a (too) low- or a (too) high-level language. A suitable MLE can be used (or developed) for each particular case. Adding domain-specific abstractions and notations increases the Closeness of Mapping between the program and the domain. The traces also help bring the prose requirements closer to the implementation code. The additional abstractions and notations are also a way of adjusting the Diffuseness/termeness of a language (or a specific program). Generally, a more terse program is better, since it exhibits lower complexity [24], assuming the language constructs used to achieve the terseness are known to all involved parties. Finally, as we have discussed above, using the right abstractions reduces the Error-proneness of programs because programmers do not have to deal with low-level details irrelevant for the problem at hand.

Learning the MLEs In order to use the MLEs effectively, users have to learn them. This cannot be avoided. However, as a consequence of the ubiquitous IDE support available in MPS, learning the MLEs is relatively simple. We also feel that learning the MLEs is a worthwhile price to pay for the security benefits. As discussed in [25], the usability and learnability of the projectional editor are appropriate for most end users.

Developing the MLEs The effort of developing the MLEs obviously depends on the level of sophistication of the MLE, but it is generally moderate. For example, the trysquentially MLE (including syntax, type system, transformation and IDE support) can be developed in one hour by an experienced MPS language developer. Developing the tabular notation for an existing state machine language takes less than a day. The reason for the low efforts is that language workbenches such as MPS are optimized for rapid development of languages (this is discussed in [14]). The modular nature of the MLEs makes the overall complexity manageable. Modularity also allows growing the language [26] over time, developing extensions only as the need arises. This is also how the mbeddr C extensions were developed.

Trusting the MLEs When we use higher-level extensions of a language in order to abstract over “irrelevant” details we implicitly trust the extension in two ways. First, we trust that we understand the MLE well enough for us to use it correctly. A well-defined extension should be relatively obvious to the users, so the risk of “using it wrong” is low (but not zero). Second, we trust the transformation that maps the MLE to its equivalent base language implementation. This is an example of tool qualification [27] in the sense through some mechanism we have to build trust that the semantics of the MLE is correct. In practice, this is done via (a sufficiently large) set of test cases as well as based on experience in practice (“proven in use” in ISO 26262). Although this may be sufficient in some use cases, other use cases require the correctness of the transformation to be proven by ana-

IV. Discussion

mbeddr’s approach to improving security relies on domain-specific extensions to C programs. We have demonstrated the advantages and opportunities of this approach in the previous section. In this section we critically discuss the approach.

Evaluating the MLEs Whether MLEs actually improve security can only be shown by experience, systematic attempts at exploiting the systems, or systematic code review. None of this has been done. In this paragraph we make two arguments why MLEs are a promising direction nonetheless. First, the experience gathered with mbeddr’s extensions have shown to improve modularity, testability and robustness of embedded software [14], [12]. A completely verified pacemaker implementation is discussed in [13]. We argue that improved robustness is an important building block of software security, since robust software has a reduced attack surface.

Second, we argue that the MLEs make C generally a better language according to Green’s Cognitive Dimensions of Notations [23], a set of established language evaluation criteria2. Table 5 contains the dimensions most

\[
\begin{align*}
\text{exported component} & \quad \text{Judge2 extends nothing} \{
\begin{align*}
\text{int16 points} &= 0; \\
\text{void judge\_reset()} &\iff \text{op judge\_reset} \{
\begin{align*}
\text{points} &= 0;
\end{align*}
\end{align*}
\}
\end{align*}
\]

Fig. 6. A piece of code can be annotated with a review state; yellow refers to ready for review, green means reviewed. The review state is persistent and survives diff/merge operations.

Fig. 7. An mbeddr assessment is used to collect the information about the review state of the various reviewable parts of the system.
lyzing the transformations. Eelco Visser and his group are working on using more formal, more analyzable languages for defining languages [28]. A related issue is known as feature interaction [29]: currently there is no way of predicting what happens if several independent MLEs are combined in the same program. Structurally and syntactically it is never a problem (thanks to projectional editing). But semantic interactions cannot be predicted because there is no formal description of the semantics. However, in practice this problem has not occurred with the 50+ C extensions developed in mbddr.

**Tool Lock-in** mbddr, as well as the MLEs developed and suggested for improving software security require the use of the MPS language workbench (for developing the MLEs and also for writing code). At this point there is no way this can be avoided; there are no interoperability standards for language workbenches. However, both MPS and mbddr are open-source software.

**Other Languages and Tools** In this paper we focus on C because a lot of secure software (in embedded and cyber-physical systems, the Internet of Things, as well as in operating systems and web servers) is written in C. However, the approach can also be used with other languages and tools. For example, MPS ships with an extensible version of Java; similar MLEs can be developed. The approach can also be used with other language workbenches. In particular, Spoofox [30] and Rascal [31] support some of the same language extension facilities as MPS (even though they are not projectional editors and hence are not as flexible regarding the notations).

V. Summary

We have shown how modular language extension, in combination with the infrastructure provided by mbddr and MPS, can be used to improve the security of embedded software. While empirical evaluation is still pending, we have argued why we consider the approach promising. Our future work includes the development of specific security-relevant MLEs, as well as their systematic evaluation. We are convinced that MPS and mbddr are useful platforms for research on improving software security through language engineering and we encourage other research groups to experiment with it.

References


