Towards Improving Software Security using Language Engineering and mbeddr C

Markus Voelter
independent/itemis
voelter@acm.org

Zaur Molotnikov
fortiss
molotnikov@fortiss.org

Bernd Kolb
itemis AG
kolb@itemis.de

Abstract

This paper explores the use of domain-specific languages for improving software security, which deals with developing software in a way that is not maliciously exploitable. Specifically we demonstrate how modular extension of the C programming language can help with technical and process-related aspects of software security. Some of these examples are already implemented, some are analytical extrapolations from related work we have done in the past; a detailed empirical evaluation has not yet been done. We rely on mbeddr, an extensible version of C developed with the JetBrains MPS language workbench. We conclude the paper with a discussion of the potential drawbacks of the approach and how these can be addressed in the future.

1. Introduction and Contribution

Software security refers to the security properties of a software system's implementation [1]. Various programming techniques as well as process practices can help with building a secure implementation. Many of the software security weaknesses 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 security (and/or safety) flaws can cost a lot of money, expose networks, damage physical systems or endanger lives (examples of attacking an aircraft can be found in [3]). Hence, addressing potential security problems in C-based software is of paramount importance for these systems. We claim in this paper that it is possible to improve over the general purpose languages (and in particular, C) by transitioning to new, domain-specific tools and languages. Language engineering [4], the notion of building, extending and composing languages, makes developing such languages and tools feasible. Language engineering relies on language workbenches [5, 6], which are a class of tools that makes language implementation efficient.

Contribution This paper demonstrates security-enhancing language extensions for C. Some are implemented, others are analytical extrapolations of previous work. Based on these extensions, we proceed to pointing out the directions for future research.

2. Language Engineering, MPS and mbeddr

Language Engineering and MPS Language engineering refers to building, extending and composing languages. The field encompasses general-purpose programming languages and domain-specific languages (DSLs) [4]. Language workbenches [5, 6] are tools for efficiently designing and implementing languages. JetBrains Meta Programming System (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 concrete syntax of a language to construct 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 MPS can work with a wide variety of (unparsable) notations such as mathematical symbols, tables and diagrams [8]. Since a projectional editor never encounters grammar ambiguities, they can support language composition [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; the study in [10] shows that users are mostly agreeable with the editor after a short while of getting used to it.

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 a future controls engineering product by Siemens PLM Software.

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.

3. Language Engineering and Security

Developing secure software relies on techniques and process. Techniques (Section 3.1) 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 a problematic 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 (Section 3.2) 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. We now explore the potential benefits of language engineering, MPS and mbeddr for both aspects.

3.1 Techniques

In this subsection we describe how extensible languages and language workbenches can be utilized to enhance software security by, for example, adding additional markup and checks to the source code, using higher level abstractions that prevent users from getting low-level details wrong or by changing the semantics of existing constructs in a way that makes the binary more secure.

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:

```
int16/m/ dAlt = cur->alt - prev->alt;
int8/s/ dTime = cur->time - prev->time;
        Error: type int16 /s · m^(-1)/ is not a subtype of int16 /mps/
cur->vSpeed = dTime / dAlt;
```

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_n and P_s). For security, it is crucial that no unencrypted data is in the non-secure part ($d_u \notin P_n$) and that data is encrypted as it moves from P_s to P_n. A set of annotations on types, variables and modules similar to physical units 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. An example for a security-relevant MLE is the `trysequentially` statement (which 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/`).

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

`trysequentially` invokes a sequence of functions, each returning 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:

```
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. Since the idiomatic C code is automatically generated from the more intentional and less error-prone `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.

We find another example to apply MLEs in Amazon's s2n TLS library (`https://github.com/awslabs/s2n`) They use macros like the following one extensively:

```
#define GUARD(x) if ((x)<0) return -1
```

The problems here are typical for macros. For example, the definition can be changed externally and it will not be prevented by an IDE: `GUARD( do_something ) + 1;` will return 0 in the case of failure. Using a language extension instead of a macro, an IDE can restrict the way the extension is used and prevent such problems.

The second problem of this macro is typical for macros that contain a `return` statement. Before returning from a function one should deallocate all the resources allocated in the function. However, the `return` statement in the macro can lead to dangling resources. Using language engineering, this problem can be solved in various ways. One could prevent the use of `return` statements in macros, or one could require to put resource deallocation code into a separate block which is then generated to be executed with every return statement (similar semantics to smart pointers).

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

```
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 on the stack still contains the actual data and can potentially be exploited. To avoid this, the semantics of C should be changed in the following way: memory used by local variables that leave their block should automatically be zeroed. This can be achieved easily by a transformation that inserts zeroing 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 to avoid unacceptable performance overhead.

A related feature is the prevention of writing this variable to disk as part of paging. Operating systems provide APIs to mark memory areas so as to prevent them from being paged. The code generator can call these APIs for all variables marked as `secure`.

Exploiting the Generation Step Most language workbenches 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.

A common attack vector are side-channel attacks [18] which exploit non-functional properties of a system to reverse-engineer details about the program's implementation or about key material. A timing side-channel attack exploits timing properties. 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.

An alternative way of decoupling execution time from input is to ensure that the respective part of the program always runs for the same time, for every possible valid input. This requirement is expressed through naming conventions in Amazon's s2n library as shown below, and developers ensure the requirement manually.

```
/* Returns 1 if a and b are equal, in constant time */
int s2n_constant_time_equals(string, string, len);
```

Encoding this information in the name prevents the IDE from ensuring that the function actually runs in constant time. Using MLEs, an annotation can mark functions as `constant time`. Static analysis can ensure that every path through the program has the same execution time. A less sophisticated solution measures the elapsed time for any particular execution and then busy-waits to extend the time to the required constant time as necessary.

Another example of exploiting the generation step is the introduction of additional runtime checks. 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 and the subsequent crash of the application.

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 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] or Java Pathfinder [20]. 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

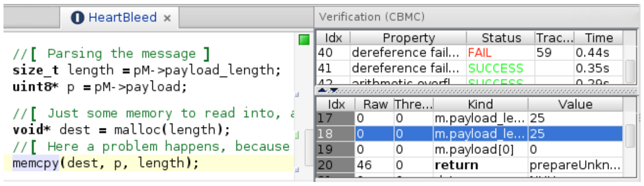

Figure 1. Running the verifier to find the Heartbleed problem in mbeddr C.

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 how to verify the functional safety of a pacemaker implementation. 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, which wrongly assumes the well-formedness of packets received from the network. Below we define a simplified heartbeat message data type:

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

Unfortunately, this is not legal C: it is not possible to specify the array length via a member of the same struct. Instead, the struct must use a pointer to the actual data. OpenSSL's mistakes was trusting the `payload_length` value, reading beyond the end of the buffer referred to by the pointer. Such buffer overruns represent a serious security vulnerability. This problem can be detected using verification. To enable it, we create a message with a nondeterministically assigned data buffer, meaning we instruct CBMC to assume all possible values for the variable. To achieve this, we use the `assign nondet` MLE, specifically built for CBMC-based verifications:

```
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 nondeterministic assignment in `prepareUntrustedMessage()` results in 1 byte allocated in the payload, and the length set to 25. The effort to implement this verification is low, assuming the MLEs for CBMC-based verification are available; only the `prepareUntrustedMessage()` and a call to the verified dispatching function are required.

MLEs 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. Such a data type recently been added to mbeddr.

Finally, another area where first-class extensions can be combined with verification is the specification of communication protocols, which are a major vector for attacks [21, 22]. If they are expressed as tables, state machines or sequence diagrams, this helps users to visually detect invalid states. In addition, model checking techniques, as discussed above, have also been used successfully to verify the correctness of communication protocols [23, 24].

3.2 Process

In this subsection we discuss how better abstraction, domain-specific notations and review support enable a more secure development process. The stakeholders for the process aspects below are mainly the developers themselves; only for the audits discussed at the end of this subsection do we expect non-developers to become involved.

Better Abstraction, Simplified Review Good abstractions 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; a graphical notation is also available in mbeddr. Another example for an easily-readable notation is given in Fig. 3.

Tracing Code reviews are done to ensure the correctness of the code (verification), but also to establish the code's corre-

[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.

```
statemachine TrafficLights initial = red {
  in event timer(int64 t) <no binding>
  in event buttonPressed() <no binding>
  var boolean button = false
  var int64 tEnter = 0
  state red {
    on buttonPressed [ ] -> red { button = true; }
    on timer [t - tEnter > 1000] -> green { tEnter = t; }
  } state red
  state green {
    on timer [t - tEnter > 500] -> red { tEnter = t; }
  } state green
}
```

```
statemachine TrafficLights initial = red {
```

		Events	
		timer(int64 t)	buttonPressed()
States	red	[t - tEnter > 1000] -> green { tEnter = t; }	[ ] -> red { button = true; }
	green	[t - tEnter > 500] -> red { tEnter = t; }	

```
}
```

Figure 2. A state machine edited as text and as a table.

spondence 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 [25] 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 [26] (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 [26]. 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

```
int32 averageIntArray(int32[] arr, int32 size) {
  return
```

$$\frac{\sum_{i=0}^{size} arr[i]}{size};$$

```
} averageIntArray (function)
```

Figure 3. Mathematical symbols used in C code simplify review of algorithmic code.

```
[#constant LANDING = 100;] -> implements FullStop
exported statemachine FlightAnalyzer initial = beforeFlight {
  state beforeFlight {
    on next [tp->alt > 0 m] -> airborne
    [exit { points += TAKEOFF; }] -> implements PointsForTakeoff
  } state beforeFlight
```

Figure 4. The green-shaded labels are requirements traces. They can be attached to any program node, here they are attached to C constants and a state machine exit action.

vector), or that every security requirement has at least one trace. An example of an assessment is shown in Fig. 6.

Code Review and Security Audit mbeddr supports tracking the review state of code. This can be done at a customizable granularity, and for code expressed in any language. Code starts out as *not reviewed*. It can then be marked as *ready for review* (yellow; see Fig. 5). Once reviewed, the state changes to *reviewed* (green; the color scheme is based on [27]). Upon the change to yellow or green, a hash of the code structure is created and stored with the code itself (in an annotation that is optionally visible to the user). An assessment can be used to get an overview of the review state of the different parts of a system (an example is shown in Fig. 6). When the assessment is updated, the hashes are recalculated to determine which parts have changed and must be reviewed again. Code that has been modified since the last review is marked as *raw* (red).

This facility can be used for regular, team-internal code reviews that aim at detecting bad practices, potential for reuse, convoluted algorithms or bad naming. However, the same approach can also be used for security audits. Compared to code reviews, these are typically performed by different people and have a different goal: finding security vulnerabilities. They often go deeper, and should ideally be performed after every change to the code base, and only on the parts that changed (plus the locations affected by this change, which can be found through data flow and other analyses). The facilities discussed in the above paragraph can detect such changed pieces of code. To ensure that the code has been audited by the external team, the hash used for detecting the changes can be signed with the private key of the auditors. This way it can be cryptographically ensured

```
[Review: ready   Reviewed 6 days ago by zaur ]
exported component Judge2 extends nothing {
  int16 points = 0;
  void judger_reset() <= op judger.reset {
    points = 0;
} }
```

Figure 5. A piece of code can be annotated with a review state. The colors have the following meaning: yellow is *ready for review*, green is *reviewed*, red is *recently created, raw*. The review state is persistent and survives diff/merge operations.

```
Assessment: ReviewOfComponentsStuff
query:      code review summary for chunk Components

| reviewed         | instancesJudging [InstanceConfiguration]
| ready            | Judge2 [AtomicComponent]
| raw              | testJudging [TestCase]
| not reviewed yet | ContractMessages [MessageDefinitionTable]
```

Figure 6. An mbeddr assessment is used to collect the information about the review state of the various reviewable parts of the system.

that the review has been performed by those authorized to do the review.

4. Discussion

mbeddr's approach to improving security relies on domain-specific extensions to C programs. We have demonstrated the potential 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 [12, 14]. 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 [28], a set of established language evaluation criteria[2]. The table in Fig. 7 contains the dimensions most 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

[2] Even though it is called Cognitive Dimensions of *Notations*, some of the dimensions actually apply to the language and its abstractions, not just the notation (concrete syntax).

Abstraction Gradient	What are the minimum and maximum levels of abstraction exposed by the notation? Encapsulation?
Closeness of Mapping	How closely does the notation correspond to the problem world?
Diffuseness/Terseness	How many symbols or how much space does the notation require to produce a certain result or express a meaning?
Error-proneness	To what extent does the notation influence the likelihood of mistakes?

Figure 7. Relevant Cognitive Dimensions of Notations.

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/terseness* of a language (or a specific program). Generally, a more terse program is better, since it exhibits lower complexity [29], 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. These are all the reasons why we believe that MLEs have a good potential in secure software development.

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 [30], the usability and learnability of the projectional editor are appropriate for most end users. Of course, to make this practical, training material on the MLEs and the concepts behind them must be provided.

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 `trysequentially` MLE (including syntax, type system, transformation and IDE support) can be developed in one hour by an experienced MPS language engineer. 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 [31] over time, developing extensions only as the need arises, avoiding costly up-front investments into MLEs that might not in fact be needed.

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 [32] 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, others require the correctness of the transformation to be proven by analyzing the transformations. Eelco Visser and his group are working on using more formal, more analyzable languages for defining languages [33]. A related issue is known as feature interaction [34]: 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 mbeddr.

In addition, the implementation of MLEs itself now becomes a critical part of the secure software. If an attacker or a careless programmer changes the language or the transformations he could break the security of the software implemented in the language without changing the software code itself. Thus MLEs should potentially be a part of the software project. They should be a subject for audits and reviews, changes to them should be tracked.

Working with Legacy Code Existing C code has not been written using the facilities discussed in this paper, but it may make sense to add the extensions retroactively. It is possible to import existing code into mbeddr. Once the code is in mbeddr, it can be refactored towards the MLEs. Our future work includes an investigation into whether such refactorings can be (partially) automated.

Tool Lock-in mbeddr, 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 mbeddr are open-source software.

Other Languages 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.

Other Tools The approach can also be used with other language workbenches: Spoofax [35] and Rascal [36] support some of the same language extension facilities as MPS (they are not projectional editors and hence are not as flexible regarding the notations).

5. Related Work

Modular C extensions have been developed for mbeddr [37] and in Cox [38] (without IDE support in the latter). Similarly, language extensions for improving security are an established idea [39], and so are dedicated DSLs for specifying aspects of software security [40]. Using static analyses to verify security-relevant properties of C has also been done before [41]. What our contribution adds is the *modular* language *and* IDE extensibility for security-relevant extensions, the use of integrated non-textual notations, as well as the combination of language extension and static analysis.

6. Summary

We have shown how modular language extension, in combination with the infrastructure provided by mbeddr 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. Future work includes the development of specific security-relevant MLEs, as well as their systematic evaluation. We are convinced that MPS and mbeddr are useful platforms for research on improving software security through language engineering and we encourage other research groups to experiment with it.

References

[1] G. McGraw, *Software Security: Building Security In*. Addison-Wesley, 2006.

[2] R. C. Seacord, *Secure Coding in C and C++*. Pearson Ed., 2005.

[3] H. Teso, "Aircraft hacking," in *HITB Security Conference, Amsterdam, The Netherlands*, 2013.

[4] M. Voelter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth, *DSL Engineering*. dslbook.org, 2013.

[5] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?" 2005.

[6] S. Erdweg, T. Storm, M. Völter *et al.*, "The state of the art in language workbenches," in *Software Language Engineering*, ser. LNCS, M. Erwig, R. Paige, and E. Wyk, Eds. Springer, 2013, vol. 8225.

[7] "JetBrains Meta Programming System," http://www.jetbrains.com/mps/, accessed 14/9/2015.

[8] M. Voelter and S. Lisson, "Supporting diverse notations in mps' projectional editor," *GEMOC Workshop 2014*, p. 7, 2014.

[9] M. Voelter, "Language and IDE Development, Modularization and Composition with MPS," in *GTTSE 2011*, ser. LNCS. Springer, 2011.

[10] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *Software Language Engineering*. Springer, 2014, pp. 41–61.

[11] M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz, "mbeddr: instantiating a language workbench in the embedded software domain," *Automated Software Engineering*, vol. 20, no. 3, pp. 1–52, 2013.

[12] M. Voelter, "Preliminary experience of using mbeddr," in *10th Dagstuhl Workshop on Model-based Development of Embedded Systems*, 2014, p. 10.

[13] Z. Molotnikov, M. Voelter, and D. Ratiu, "Automated Domain-Specific C Verification with mbeddr," in *29th Intl. Conf. on Automated Software Engineering (ASE 2014)*, 2014, pp. 539–550.

[14] M. Voelter, "Generic tools, specific languages," dissertation, TU Delft, 2014.

[15] F. Campagne, *The MPS Language Workbench*. CreateSpace, 2014.

[16] J. Gregoire, K. Buyens, B. D. Win, R. Scandariato, and W. Joosen, "On the secure software development process: Clasp and sdl compared," in *Proc. of the 3rd Intl. Workshop on Software Engineering for Secure Systems*. IEEE CS, 2007.

[17] M. C. O. M. I. Board, *Mars Climate Orbiter Mishap Investigation Board: Phase I Report*. Jet Propulsion Laboratory, 1999.

[18] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," in *Computer Security-ESORICS 98*. Springer, 1998, pp. 97–110.

[19] E. Clarke, D. Kroening, and F. Lerda, " A Tool for Checking ANSI-C Programs ," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 2988. Springer, 2004, pp. 168–176.

[20] K. Havelund and T. Pressburger, "Model Checking JAVA Programs using JAVA PathFinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[21] D. Wagner, B. Schneier *et al.*, "Analysis of the ssl 3.0 protocol," in *The Second USENIX Workshop on Electronic Commerce Proceedings*, 1996, pp. 29–40.

[22] D. Geneiatakis, T. Dagiuklas, G. Kambourakis, C. Lambrinoudakis, S. Gritzalis, S. Ehlert, D. Sisalem *et al.*, "Survey of security vulnerabilities in session initiation protocol." *IEEE Communications Surveys and Tutorials*, vol. 8, no. 1-4, pp. 68–81, 2006.

[23] M. Kwiatkowska, G. Norman, and J. Sproston, *Probabilistic model checking of the IEEE 802.11 wireless local area network protocol*. Springer, 2002.

[24] M. Musuvathi, D. R. Engler *et al.*, "Model checking large network protocol implementations." in *NSDI*, vol. 4. Citeseer, 2004.

[25] M. Jarke, "Requirements tracing," *Commun. ACM*, vol. 41, no. 12, pp. 32–36, Dec. 1998. [Online]. Available: http://doi.acm.org/10.1145/290133.290145

[26] M. Voelter, D. Ratiu, and F. Tomassetti, "Requirements as first-class citizens: Integrating requirements closely with implementation artifacts," in *ACESMB@MoDELS*, 2013.

[27] F. Deissenboeck, M. Pizka, and T. Seifert, "Tool support for continuous quality assessment," in *13th International Workshop on Software Technology and Engineering Practice*, 2005, pp. 127–136.

[28] T. R. Green, "Cognitive dimensions of notations," *People and computers V*, pp. 443–460, 1989.

[29] J. Graylin, J. E. Hale, R. K. Smith, H. David, N. A. Kraft, W. Charles *et al.*, "Cyclomatic complexity and lines of code: empirical evidence of a stable linear relationship," *Journal of Software Engineering and Applications*, vol. 2, no. 03, p. 137, 2009.

[30] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *7th International Conference on Software Language Engineering (SLE)*, 2014.

[31] G. L. Steele, "Growing a language," *Higher-Order and Symbolic Computation*, vol. 12, no. 3, pp. 221–236, 1999.

[32] M. Conrad, G. Sandmann, and P. Munier, "Software tool qualification according to ISO 26262," SAE, Tech. Rep., 2011.

[33] E. Visser, G. Wachsmuth, A. Tolmach, P. Neron, V. Vergu, A. Passalaqua, and G. Konat, "A language designer's workbench. a one-stop-shop for implementation and verification of language designs," in *Proceedings of SPLASH 2014, Onward*, 2014.

[34] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature interaction: a critical review and considered forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[35] L. Kats and E. Visser, "The Spoofax language workbench: rules for declarative specification of languages and IDEs," in *ACM Sigplan Notices*, vol. 45, no. 10. ACM, 2010, pp. 444–463.

[36] P. Klint, T. Van Der Storm, and J. Vinju, "EASY Metaprogramming with Rascal," in *Generative and Transformational Techniques in Software Engineering III*. Springer, 2011, pp. 222–289.

[37] M. Voelter, D. Ratiu, B. Kolb, and B. Schätz, "mbeddr: instantiating a language workbench in the embedded software domain," *Autom. Softw. Eng.*, vol. 20, no. 3, pp. 339–390, 2013.

[38] R. Cox, T. Bergan, A. T. Clements, M. F. Kaashoek, and E. Kohler, "Xoc, an extension-oriented compiler for systems programming," in *ASPLOS 2008*, 2008.

[39] K. Ashcraft and D. Engler, "Using programmer-written compiler extensions to catch security holes," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, 2002, pp. 143–159.

[40] J. DeTreville, "Binder, a logic-based security language," in *Security and Privacy, 2002. Proceedings. 2002 IEEE Symposium on*, 2002.

[41] M. Aizatulin, F. Dupressoir, A. D. Gordon, and J. Jürjens, "Verifying cryptographic code in C: Some experience and the csec challenge," in *Formal Aspects of Security and Trust*. Springer, 2012, pp. 1–20.